ILOG JViews 5.5

Graph Layout User's Manual

December 2002

© Copyright 2002 by ILOG

This document and the software described in this document are the property of ILOG and are protected as ILOG trade secrets. They are furnished under a license or non-disclosure agreement, and may be used or copied only within the terms of such license or non-disclosure agreement.
No part of this work may be reproduced or disseminated in any form or by any means, without the prior written permission of ILOG S.A.

Printed in France

Table of Contents

Preface **About This Manual** vii

Chapter 1 **Introducing the Graph Layout Module** 1

What is the Graph Layout Module of ILOG JViews? 2

Composition of the ILOG JViews Graph Layout Module 2

The ILOG JViews Graph Layout Algorithms 4

Features of the ILOG JViews Graph Layout Module 8

ILOG JViews Graph Layout Module in User Interface Applications 10

Chapter 2 **Basic Concepts** 13

Graph Layout: A Brief Introduction 14

Graph Layout in ILOG JViews 16

The Base Class: IlvGraphLayout 17

Basic Operations with IlvGraphLayout 17

Layout Parameters and Features in IlvGraphLayout 20

Chapter 3 **Getting Started with Graph Layout** 37

Basic Steps for Using Layout Algorithms: A Summary 38

Sample Java Application 38

Chapter 4 **Layout Algorithms** 43

Determining the Appropriate Layout Algorithm 44

Typical Ways for Choosing the Layout 49

Generic Features and Parameters Support 51
Layout Characteristics 52
Topological Mesh Layout (TML) 54
Spring Embedder Layout 72
Uniform Length Edges Layout 81
Tree Layout 94
Hierarchical Layout 134
Link Layout 185
Random Layout 217
Bus Layout 222
Circular Layout 238
Grid Layout 255
Recursive Layout 265
Multiple Layout 282

Chapter 5 Using Advanced Features 291

Using a Layout Report 292
Using Event Listeners 294
Redrawing the Grapher after Layout 296
Using the Graph Model 297
Laying Out a Non-JViews Grapher 303
Laying Out Connected Components of a Disconnected Graph 304
Laying Out Nested Graphs 306
Saving Layout Parameters and Preferred Layouts 315
Using the Filtering Features to Lay Out a Part of an IlvGrapher 321
Choosing the Layout Coordinate Space 323
Releasing Resources Used During the Layout 328
Defining a New Type of Layout 329
Questions and Answers about Using the Layout Algorithms 333

Chapter 6 Automatic Label Placement 339

Getting Started with Labeling 340

The Base Classes: IlvLabelingModel and IlvLabelLayout 346
The Annealing Label Layout 354
The Random Label Layout 376
Using Advanced Features 379
Defining Your Own Labeling Model 384

Chapter 7 Using Graph Layout Beans 391

Graph Layout Classes Available as Beans 392
Creating a Simple Applet Using ILOG JViews Graph Layout Beans 393

Glossary 411

Index 419

About This Manual

The ILOG JViews Component Suite provides special support for applications that need to display graphs, or networks, of nodes and links. Any graphic object can be defined to behave like a node and can be connected to other nodes via links, which themselves can have many different forms. Used in conjunction with layout algorithms, the ILOG JViews grapher is often used to display network topologies for telecommunications networks and systems management applications.

What Is in This Manual

The ILOG JViews Graph Layout module provides high-level, ready-to-use graph drawing services that allow you to obtain readable representations easily.

This manual contains the following chapters:

- Chapter 1, *Introducing the Graph Layout Module* describes the Graph Layout module of ILOG JViews and its features.
- Chapter 2, *Basic Concepts* provides background information and basic concepts for using Graph Layout.
- Chapter 3, *Getting Started with Graph Layout* provides information to get started quickly using Graph Layout.

- Chapter 4, *Layout Algorithms* describes the layout algorithms provided with the Graph Layout module.
- Chapter 5, *Using Advanced Features* provides information on using a layout report, using layout event listeners, using a graph model, laying out a non-JViews grapher, layout out disconnected graphs, saving layout parameters to a file, laying out a portion of a graph, laying out graphs with nonzoomable objects, and defining new types of layouts.
- Chapter 6, *Automatic Label Placement* describes the label layout facilities provided with the Graph Layout module.
- Chapter 7, *Using Graph Layout Beans* shows you how to use the ILOG JViews Beans and the Graph Layout Beans when creating an applet within an Integrated Development Environment (IDE).

At the end of the manual you will find a *Glossary* containing definitions of the basic technical terms used in this manual.

Related Documentation

The following documentation may provide helpful information when using ILOG JViews Graph Layout.

Books

The first book dedicated to graph layout has been published:

Di Battista, Giuseppe, Peter Eades, Roberto Tammassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice Hall, 1999 (see `http://www.cs.brown.edu/people/rt/gdbook.html` or `http://www.prenhall.com/books/esm_0133016153.html`).

Graph layout is closely related to graph theory, for which extensive literature exists. See:

Clark, John and Derek Allan Holton. *A First Look at Graph Theory.* World Scientific Publishing Company, 1991.

For a mathematics-oriented introduction to graph theory, see:

Diestel,Reinhard, *Graph Theory,* 2nd ed. Springer-Verlag, 2000.

A more algorithmic approach may be found in:

Gibbons, Alan. *Algorithmic Graph Theory.* Cambridge University Press, 1985.

Gondran, Michel and Michel Minoux. *Graphes et algorithmes*, 3rd ed. Eyrolles, Paris, 1995 (in French).

Bibliographies

A comprehensive bibliographic database of papers in computational geometry (including graph layout) can be found:

The Geometry Literature Database
(`http://compgeom.cs.uiuc.edu/~jeffe/compgeom/biblios.html`)

The recommended bibliographic survey paper is the following:

Di Battista, Giuseppe, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. "Algorithms for Drawing Graphs: an Annotated Bibliography." *Computational Geometry: Theory and Applications* 4 (1994): 235-282 (also available at `http://www.cs.brown.edu/people/rt/gd-biblio.html`).

Journals

The following are electronic journals:

Journal of Graph Algorithms and Applications
(`http://www.cs.brown.edu/publications/jgaa`)

Algorithmica
(`http://link.springer-ny.com/link/service/journals/00453/`)

Computational Geometry: Theory and Applications
(`http://www.elsevier.nl/inca/publications/store/5/0/5/6/2/9/`)

Journal of Visual Languages and Computing
(`http://www.academicpress.com/jvlc`)

The following journals occasionally publish papers on graph layout:

Information Processing Letters
(`http://www.elsevier.nl/inca/publications/store/5/0/5/6/1/2/`)

Computer-aided Design
(`http://www.elsevier.nl/inca/publications/store/3/0/4/0/2/`)

IEEE Transactions on Software Engineering
(`http://www.computer.org/tse/`)

Many papers are presented at conferences in Combinatorics and Computer Science.

Conferences

An annual Symposium on Graph Drawing has been held since 1992. The proceedings are published by Springer-Verlag in the *Lecture Notes in Computer Science* series. For the 2001 edition, see
`http://link.springer.de/link/service/series/0558/tocs/t2265.htm`. For the 2003 Symposium, to be held in Perugia, Italy, see `http://www.gd2003.org`.

C H A P T E R

1

Introducing the Graph Layout Module

This chapter introduces you to the Graph Layout module of ILOG JViews. The following topics are covered:

- *What is the Graph Layout Module of ILOG JViews?*
- *Composition of the ILOG JViews Graph Layout Module*
- *The ILOG JViews Graph Layout Algorithms*
- *Features of the ILOG JViews Graph Layout Module*
- *ILOG JViews Graph Layout Module in User Interface Applications*

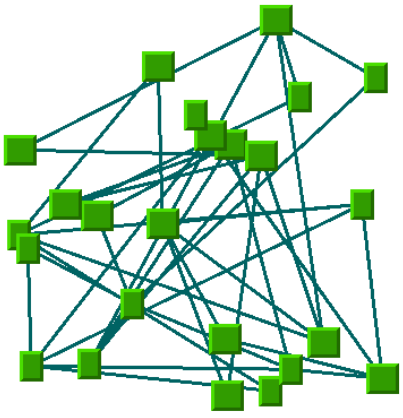
What is the Graph Layout Module of ILOG JViews?

Many types of complex business data can best be visualized as a set of nodes and interconnecting links, more commonly called a graph or a network. Examples of graphs include business organizational charts, workflow diagrams, telecom network displays, and genealogical trees. Whenever these graphs become large or heavily interconnected, it becomes difficult to see the relationships between the various nodes and links (the “edges”). This is where ILOG JViews Graph Layout algorithms help.

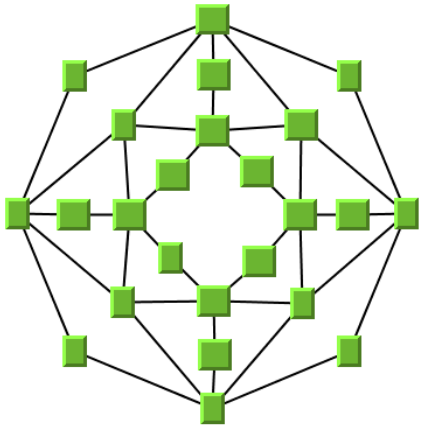
The ILOG JViews Graph Layout module provides high-level, ready-to-use relationship visualization services. It allows you to take any “messy” graph and apply a sophisticated graph layout algorithm to rearrange the positions of the nodes and links. The result is a more readable and understandable picture.

Take a look at two sample drawings of the same graph.

Here no format layout algorithm was used. The nodes were placed randomly when the graph was drawn.



Using one of the layout algorithms provided in ILOG JViews, the following drawing was obtained:



In the second drawing, the layout algorithm has distributed the nodes quite uniformly, avoiding overlapping nodes and showing the symmetries of the graph. This drawing presents a much more readable layout than does the first drawing.

Composition of the ILOG JViews Graph Layout Module

The ILOG JViews Graph Layout module is composed of the following packages:

- `ilog.views.graphlayout`: A high-level, generic framework for the graph layout services provided by ILOG JViews.

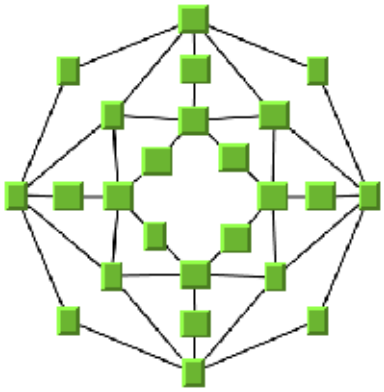
- Layout algorithm packages:
  - `ilog.views.graphlayout.bus`: A layout algorithm designed to display bus network topologies (that is, a set of nodes connected to a bus node).
  - `ilog.views.graphlayout.circular`: A layout algorithm that displays graphs representing interconnected ring and/or star network topologies.
  - `ilog.views.graphlayout.grid`: A layout algorithm that arranges the disconnected nodes of a graph in rows, in columns, or in the cells of a grid.
  - `ilog.views.graphlayout.hierarchical`: A layout algorithm that arranges nodes in horizontal or vertical levels such that the links flow in a uniform direction.
  - `ilog.views.graphlayout.link`: A layout algorithm that reshapes the links of a graph without moving the nodes.

    `ilog.views.graphlayout.link.longlink`: For long orthogonal links.
    `ilog.views.graphlayout.link.shortlink`: For short links.
  - `ilog.views.graphlayout.multiple`: A facility that combines multiple layout algorithms into one graphic object.
  - `ilog.views.graphlayout.random`: A layout algorithm that moves the nodes of the graph at randomly computed positions inside an user-defined region.
  - `ilog.views.graphlayout.recursive`: A layout algorithm that can be used to control the layout of nested graphs (containing subgraphs and intergraph links).
  - `ilog.views.graphlayout.springembedder`: A layout algorithm that can be used to lay out any type of graph.
  - `ilog.views.graphlayout.topologicalmesh`: A layout algorithm that can be used to lay out cyclic graphs.
  - `ilog.views.graphlayout.tree`: A layout algorithm that arranges the nodes of a tree horizontally or vertically, starting from the root of the tree. A radial layout mode allows you to arrange the nodes of a tree on concentric circles around the root of the tree.
  - `ilog.views.graphlayout.uniformlengthedges`: A layout algorithm that can be used to lay out any type of graph and allows you to specify the length of the links.
- `ilog.views.graphlayout.labellayout`: A layout algorithm for automatic placement of labels.
  - `ilog.views.graphlayout.labellayout.annealing`: For close label positioning.
  - `ilog.views.graphlayout.labellayout.random`: For random placement.
- `ilog.views.graphlayout.swing`: Swing components useful for creating applications mixing ILOG JViews Graph Layout and Swing.

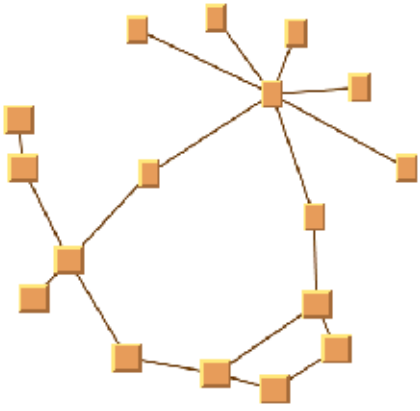
The ILOG JViews Graph Layout Algorithms

The Graph Layout module of ILOG JViews provides numerous ready-to-use layout algorithms. They are shown below with sample illustrations. In addition, new layout algorithms can be developed using the generic layout framework of the ILOG JViews Graph Layout module.

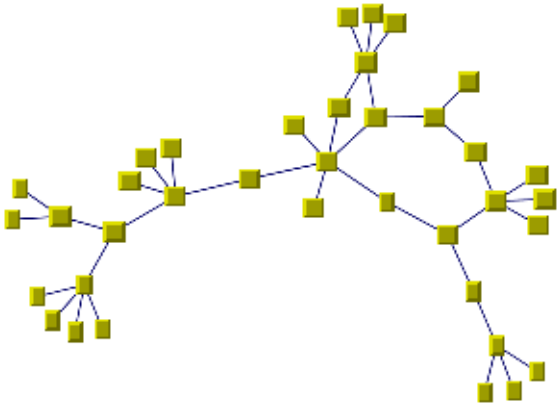
Topological Mesh Layout (TML)



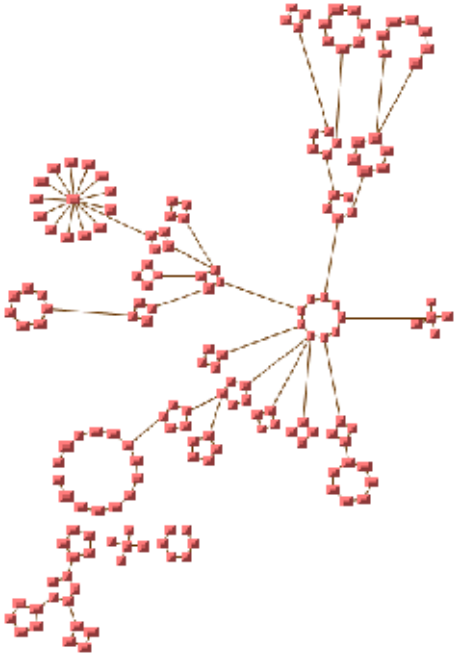
Spring Embedder Layout



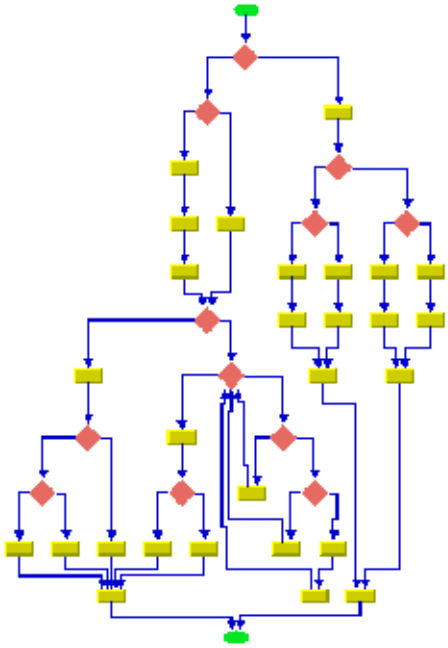
Uniform Length Edges Layout



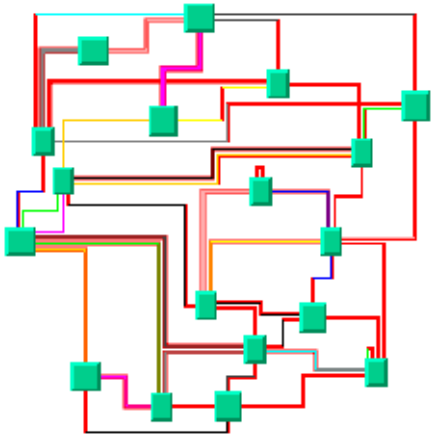
Circular Layout (Ring/Star)



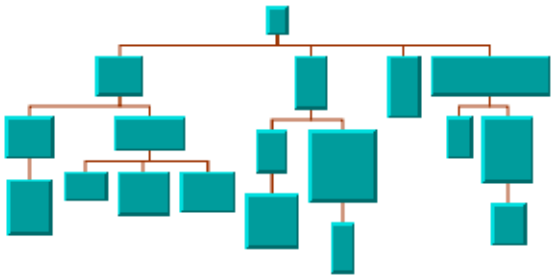
Hierarchical Layout



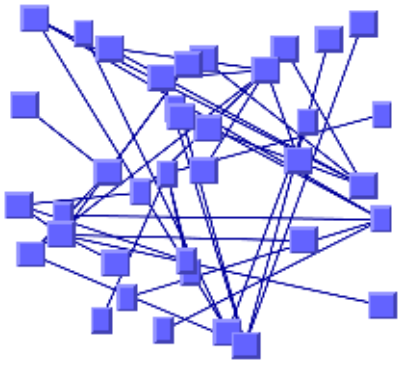
Link Layout



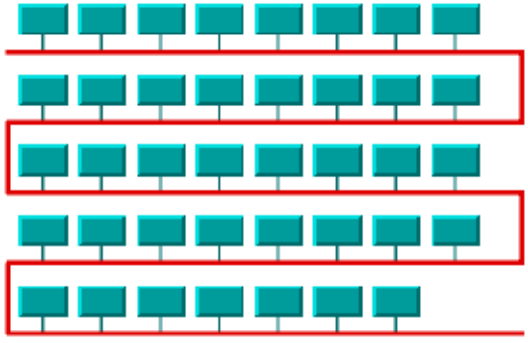
Tree Layout



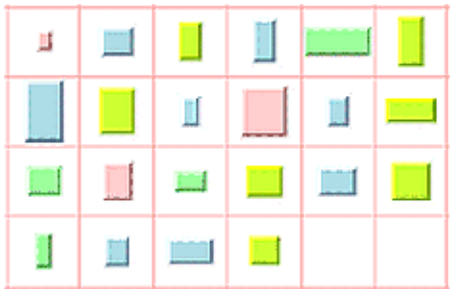
Random Layout



Bus Layout



Grid Layout



Features of the ILOG JViews Graph Layout Module

The Graph Layout module provides the following features for using the layout algorithms. (Note that some of these features are not supported by all the algorithms provided with ILOG JViews.)

- Capability to fit the layout into a manager view or a user-defined rectangle
- Capability to take into account the size of the nodes when performing the layout to avoid overlapping nodes
- Capability to perform the layout using only the nodes and links that are on user-defined layers of the graph
- Capability to perform the layout only on those parts of the graph that meet user-defined conditions
- Capability to use non-JViews graphers
- Layout reports providing information concerning the behavior of the layout algorithm
- Layout-event listeners that can receive and report information during the graph layout
- A generic framework for customizing the layout algorithms. The following generic features and parameters are defined. (Note that not all the layout algorithms provided with ILOG JViews support all these parameters. Whether a generic parameter is supported depends on the particular layout algorithm.)
  - Allowed Time

    This parameter allows the layout algorithm to stop computation when a user-defined time specification is exceeded.
  - Animation

    This parameter allows the layout algorithm to redraw the graph after each iteration or step.

- Fixed Nodes

  This parameter allows the layout algorithm to preserve the location of the specified nodes. Certain nodes can be specified as fixed and will not be moved when the layout is performed. The layout algorithm can “pin” specified nodes in place.

- Fixed Links

  This parameter allows the layout algorithm to preserve the shape of the specified links. Certain links can be specified as fixed and will not be reshaped when the layout is performed. The layout algorithm can “pin” specified links in place.

- Filtering

  The layout algorithms are able to perform the layout using only the nodes and links that are on user-defined layers of the grapher, or to exclude nodes and links on an individual basis.

- Layout of Connected Components

  This parameter allows you to automatically lay out the connected components of a disconnected graph.

- Layout Region

  This parameter allows the layout algorithm to control the size of the graph drawing.

- Percentage Completion Calculation

  This parameter allows the layout algorithm to provide an estimation of how much of the layout has been completed.

- Random Generator Seed Value

  This parameter allows the layout algorithm to use randomly generated numbers that can be initialized with a user-defined seed value. These seed values are then used during layout computations to produce different layouts of the graph.

- Save Parameters to Named Properties

  This parameter provides support for saving the layout parameters as named properties in `.ivl` files.

- Stop Immediately

  This parameter allows the layout algorithm to stop layout computation immediately when an outside event occurs.

- Use Default Parameters

  This parameter allows the layout algorithm to return to using default parameter settings after the default settings have been modified.

ILOG JViews Graph Layout Module in User Interface Applications

Many fields use graph drawings and graph layouts in user interface applications. Therefore, the ILOG JViews Graph Layout module is particularly well-suited for these kinds of applications. The following is a list of some of the fields where the graph layout capabilities of the ILOG JViews Graph Layout module can be used:

- **Telecom and Networking**
  - LAN Diagrams (*Bus Layout, Circular Layout*)
  - WAN Diagrams (*Spring Embedder, Uniform Edge Length*)
- **Electric Engineering**
  - Logic Diagrams (*Hierarchical Layout*)
  - Circuit Block Diagrams (*Hierarchical Layout, Link Layout, Bus Layout*)
- **Industrial Engineering**
  - Industrial Process Charts (*Hierarchical Layout*)
  - Schematic Design Diagrams (*Link Layout, Hierarchical Layout*)
  - Equipment/Resource Control Charts (*Bus Layout, Link Layout*)
- **Business Processing**
  - Workflow Diagrams (*Hierarchical Layout*)
  - Process Flow Diagrams (*Hierarchical Layout*)
  - Organization Charts (*Tree Layout, Circular Layout*)
  - Entity Relation Diagrams (*Link Layout*)
  - PERT Charts (*Hierarchical Layout*)
- **Software Management/Software (Re-)Engineering**
  - UML Diagrams (*Hierarchical Layout, Tree Layout*)
  - Flow Charts (*Hierarchical Layout*)
  - Data Inspector Diagrams (*Link Layout, Hierarchical Layout*)
  - Call Graphs (*Spring Embedder, Uniform Edge Length Layout, Hierarchical Layout, Tree Layout*)
- **CASE Tools**
  - Design Diagrams (*Link Layout, Hierarchical Layout*)
  - Dependency Diagrams (*Spring Embedder, Uniform Edge Length Layout*)

- **Data Base and Knowledge Engineering**
  - Semantic Networks (*Uniform Edge Length Layout, Spring Embedder, Topological Mesh Layout*)
  - Decision Trees (*Tree Layout*)
  - Database Query Graphs (*Spring Embedder, Uniform Edge Length Layout, Hierarchical Layout*)
  - Qualitative Reasoning and other Artificial Intelligence Diagrams (*Topological Mesh Layout, Spring/Uniform Edge Length Layout*)
- **The World Wide Web**
  - Web Site Maps (*Tree Layout*)
  - Web Hyperlink Neighborhood (*Spring Embedder, Uniform Edge Length Layout, Circular Layout*)

C H A P T E R

2

Basic Concepts

In this chapter, you will learn about some basic concepts and background information that will help you when using the ILOG JViews Graph Layout module. The following topics are covered:

- *Graph Layout: A Brief Introduction*
- *Graph Layout in ILOG JViews*
- *The Base Class: IlvGraphLayout*
- *Basic Operations with IlvGraphLayout*
- *Layout Parameters and Features in IlvGraphLayout*

Graph Layout: A Brief Introduction

This section provides some background information about graph layout in general, not specifically related to the ILOG JViews Graph Layout module.

Simply speaking, a graph is a data structure which represents a set of entities, called nodes, connected by a set of links. (A node can also be referred to as a vertex. A link can also be referred to as an edge or a connection.) In practical applications, graphs are frequently used to model a very wide range of things: computer networks, software program structures, project management diagrams, and so on. Graphs are powerful models because they permit applications to benefit from the results of graph theory research. For instance, efficient methods are available for finding the shortest path between two nodes, the minimum cost path, and so on.

Graph layout is used in graphical user interfaces of applications that need to display graph models. To lay out a graph means to draw the graph so that an appropriate, readable representation is produced. Essentially, this involves determining the location of the nodes and the shape of the links. For some applications, the location of the nodes may be already known (based on the geographical positions of the nodes, for example). However, for other applications, the location is not known (a pure "logical" graph) or the known location, if used, would produce an unreadable drawing of the graph. In these cases, the location of the nodes must be computed.

But what is meant by an "appropriate" drawing of a graph? In practical applications, it is often necessary for the graph drawing to respect certain quality criteria. These criteria may vary depending on the application field or on a given standard of representation. It is often difficult to speak about what a good layout consists of. Each end user may have different, subjective criteria for qualifying a layout as "good". However, one common goal exists behind all the criteria and standards: the drawing must be easy to understand and provide easy navigation through the complex structure of the graph.

These topics are pursued in the following sections:

- *What is a Good Layout?*
- *Methods for Using Layout Algorithms*

What is a Good Layout?

To deal with the various needs of different applications, many classes of graph layout algorithms have been developed. A layout algorithm addresses one or more quality criteria, depending on the type of graph and the features of the algorithm, when laying out a graph. The most common criteria are:

- Minimizing the number of link crossings
- Minimizing the total **area** of the drawing

- Minimizing the number of **bends** (in orthogonal drawings)
- Maximizing the smallest **angle** formed by consecutive incident links
- Maximizing the display of **symmetries**

How can a layout algorithm meet each of these quality criteria and standards of representation? If you look at each individual criteria, some can be met quite easily, at least for some classes of graphs. For other classes, it may be quite difficult to produce a drawing that meets the criteria. For example, minimizing the number of link crossings is relatively simple for trees (that is, graphs without cycles). However, for general graphs, minimizing the number of link crossings is a mathematical NP-complete problem (that is, with all known algorithms, the time required to perform the layout grows very fast with the size of the graph.)

Moreover, if you want to meet several criteria at the same time, an optimal solution simply may not exist with respect to each individual criteria because many of the criteria are mutually contradictory. Time-consuming trade-offs may be necessary. In addition, it is not a trivial task to assign weights to each criteria. Multicriteria optimization is, in most cases, too complex to implement and much too time-consuming. For these reasons, layout algorithms are often based on heuristics and may provide less than optimal solutions with respect to one or more of the criteria. Fortunately, in practical terms, the layout algorithms will still often provide reasonably readable drawings.

Methods for Using Layout Algorithms

Layout algorithms can be employed in a variety of ways in the various applications in which they are used. The most common ways of using an algorithm are the following:

- **Automatic layout**

  The layout algorithm does everything without any user intervention, except perhaps the choice of the layout algorithm to be used. Sometimes a set or rules can be coded to choose automatically (and dynamically) the most appropriate layout algorithm for the particular type of graph being laid out.

- **Semiautomatic layout**

  The end user is free to improve the result of the automatic layout procedure by hand. At times the end user can move and “pin” nodes at desired locations and perform the layout again. In other cases, a part of the graph is automatically set as “read-only” and the end user can modify the rest of the layout.

- **Static layout**

  The layout algorithm is completely redone (“from scratch”) each time the graph is changed.

- **Incremental layout**

  When the layout algorithm is performed a second time on a modified graph, it tries to preserve the stability of the layout as much as possible. The layout is not performed again from scratch. The layout algorithm also tries to economize CPU time by using the previous layout as an initial solution. Some layout algorithms and layout styles are incremental by nature. For others, incremental layout may be impossible.

Graph Layout in ILOG JViews

In ILOG JViews, graphs are instances of the class `IlvGrapher`. We call these instances *graphers*. The nodes, which are instances of `IlvGraphic`, and the links, which are instances of `IlvLinkImage`, "know" how to draw themselves. The nodes can have arbitrary coordinates or they can be "placed" interactively or by code. All that needs to be done to lay out the grapher in order to obtain a readable drawing is to compute and to assign appropriate coordinates for the nodes. In some cases, the shape of the links may also need to be modified. The main task of the Graph Layout module is to provide support for the operation of laying out a grapher—that is, drawing the graph.

The Graph Layout module of ILOG JViews benefits from its integration with the graph visualization and graph manipulation features of the ILOG JViews core library. This core library provides a wide range of very useful features to build powerful graphic interfaces easily:

- Predefined, extensible types of graphic objects for nodes and links
- A customizable mechanism to choose the contact points between links and nodes
- Grapher interactor classes
- Multiple views of the same grapher
- Management of multiple layers
- Support for nested graphs
- Selections management
- Events management
- Listeners on the contents of the grapher and/or on the views
- Printing facilities
- User-defined properties on nodes and links
- Input/output operations

For details on these features, see the *ILOG JViews Graphics Framework User's Manual*.

> ***Note:*** *The Graph Layout module allows you to add layout capabilities to applications that do not use the ILOG JViews grapher. For details, see Laying Out a Non-JViews Grapher on page 303.*

The Base Class: IlvGraphLayout

The `IlvGraphLayout` class is the base class for all layout algorithms. This class is an abstract class and cannot be used directly. You must use one of its subclasses (`IlvTopologicalMeshLayout`, `IlvSpringEmbedderLayout`, `IlvUniformLengthEdgesLayout`, `IlvTreeLayout`, `IlvHierarchicalLayout`, `IlvLinkLayout`, `IlvRandomLayout`, `IlvBusLayout`, `IlvCircularLayout`, `IlvGridLayout`). You can also create your own subclasses to implement other layout algorithms.

Despite the fact that only subclasses of `IlvGraphLayout` are directly used to obtain the layouts, it is still necessary to learn about this class because it contains methods that are inherited (or overridden) by the subclasses. And, of course, you will need to understand it if you subclass it yourself.

You can find more information about the class `IlvGraphLayout` in the following sections:

- *Basic Operations with IlvGraphLayout* on page 17 tells you about the basic methods you need using the subclasses of `IlvGraphLayout`.
- *Layout Parameters and Features in IlvGraphLayout* on page 20 contains the methods that are related to the customization of the layout algorithms.
- *Using Event Listeners* on page 294 tells you about the layout event listener mechanism.
- *Defining a New Type of Layout* on page 329 tells you how to implement new subclasses.

For details on `IlvGraphLayout` and other graph layout classes, see the *ILOG JViews Graph Layout Reference Manual*.

Basic Operations with IlvGraphLayout

When subclassing `IlvGraphLayout` or when using one of its subclasses, you will normally use the basic methods described in the following sections:

- *Instantiating a Subclass of IlvGraphLayout*
- *Attaching a Grapher*

- *Performing a Layout*
- *Detaching a Grapher*

Instantiating a Subclass of IlvGraphLayout

The class `IlvGraphLayout` is an abstract class. It has no constructors. You will instantiate a subclass as shown in the following example:

```
IlvLinkLayout layout = new IlvLinkLayout();
```

If you want to use the layout report that is returned by the layout algorithm, you need to declare a handle for the appropriate layout report class, as shown in the following example:

```
IlvGraphLayoutReport layoutReport;
```

For more information on the layout report, see *Using a Layout Report* on page 292.

Attaching a Grapher

You must attach the grapher before performing the layout. The following method, defined on the class `IlvGraphLayout`, allows you to specify the grapher you want to lay out:

```
void attach(IlvGrapher grapher)
```

For example:

```
...
IlvGrapher grapher = new IlvGrapher();
/* Add nodes and links to the grapher here */
layout.attach(grapher);
```

The `attach` method does nothing if the specified grapher is already attached. Otherwise, it first detaches the grapher that is already attached, if any. You can obtain the attached grapher using the method `getGrapher()`. If the grapher is attached in this way, a default graph model is created internally. For details on the graph model, see *Using the Graph Model* on page 297. The attached graph model can be obtained by:

```
IlvGraphModel graphModel = layout.getGraphModel();
```

Warning: *Notice that such an internally created model is not allowed to be attached to any other layout instance, nor to be used in any way once it has been detached from the layout instance. For details, see Using the IlvGrapherAdapter on page 302.*

After layout, when you no longer need the layout instance, you should call the `detach` method. If the `detach` method is not called, some objects may not be garbage-collected. This method also performs cleaning operations on the grapher (properties that may have been added by the layout algorithm on the grapher's objects are removed).

```
void detach()
```

Performing a Layout

The `performLayout` method starts the layout algorithm using the currently attached grapher and the current settings for the layout parameters. The method returns a report object that contains information about the behavior of the layout algorithm.

```
IlvGraphLayoutReport performLayout()
IlvGraphLayoutReport performLayout(boolean force, boolean redraw)
```

The first version of the method simply calls the second one with a `false` value for the first argument and a `true` value for the second argument. If the argument `force` is `false`, the layout algorithm first verifies whether it is necessary to perform the layout. It checks internal flags to see whether the grapher or any of the parameters have been changed since the last time the layout was successfully performed. A "change" can be any of the following:

- Nodes or links were added or removed.
- Nodes or links were moved or reshaped.
- The value of a layout parameter was modified.
- The transformer of a manager view (`IlvManagerView`) of the grapher has changed.

Users often do not want the layout to be computed again if no changes occurred. If there were no changes, the method `performLayout` returns without performing the layout. Note that if the argument `force` is passed as `true`, the verification is no longer performed.

The argument `redraw` determines whether a redraw of the graph is requested. For details, see *Redrawing the Grapher after Layout* on page 296.

The protected abstract method `layout(boolean redraw)` is then called. This means that the control is passed to the subclasses that are implementing this method. The implementation computes the layout and moves the nodes to new positions and/or reshapes the links.

The `performLayout` method returns an instance of `IlvGraphLayoutReport` (or of a subclass) that contains information about the behavior of the layout algorithm. It tells you whether the algorithm performed normally, or whether a particular, predefined case occurred. (For a more detailed description of the layout report, see *Using a Layout Report* on page 292.)

Note that the layout report that is returned can be an instance of a subclass of `IlvGraphLayoutReport` depending on the particular subclass of `IlvGraphLayout` you are using. For example, it will be an instance of `IlvTopologicalMeshLayoutReport` if you are using the class `IlvTopologicalMeshLayout`. Subclasses of `IlvGraphLayoutReport` are used to store layout algorithm-dependent information.

You must call the method `performLayout` inside a `try` block because it can throw an exception. The exception can be of the type `IlvGraphLayoutException` or one of its subclasses, `IlvInappropriateGraphException` and `IlvInappropriateLinkException`. The first indicates internal problems in the layout algorithm or an unexpected situation. The second exception indicates that a particular grapher cannot be laid out with the layout algorithm. For example, the Topological Mesh Layout cannot be used on an acyclic graph (that is, a pure tree). The third exception indicates that a particular type of link (or link connector) cannot be used for this layout. The recommended type of link is `IlvPolylineLinkImage` or `IlvSplineLinkImage` (or subclasses). For layouts that do not reshape the links by adding intermediate points, the class `IlvLinkImage` can also be used.

Detaching a Grapher

You call the `detach` method when you no longer need the layout instance of an attached grapher. If the `detach` method is not called, some objects may not be garbage-collected. This method also performs cleaning operations on the grapher (properties that may have been added by the layout algorithm on the grapher's objects are removed).

```
void detach()
```

Layout Parameters and Features in IlvGraphLayout

The `IlvGraphLayout` class defines a number of generic features and parameters. These features and parameters can be used to customize the layout algorithms.

Although the `IlvGraphLayout` class defines the generic parameters, it does not control how they are used by its subclasses. Each layout algorithm (that is, each subclass of `IlvGraphLayout`) supports a subset of the generic features and determines the way in which it uses the generic parameters. When you create your own layout algorithm by subclassing `IlvGraphLayout`, you decide whether you want to use the features and the way in which you are going to use them.

The `IlvGraphLayout` class defines the following generic features:

- *Allowed Time*
- *Animation*
- *Automatic Layout*
- *Coordinates Mode*
- *Layout of Connected Components*
- *Layout Region*

- *Link Clipping*
- *Link Connection Box*
- *Memory Savings*
- *Percentage Completion Calculation*
- *Preserve Fixed Links*
- *Preserve Fixed Nodes*
- *Random Generator Seed Value*
- *Save Parameters to Named Properties*
- *Stop Immediately*
- *Use Default Parameters*

Table 4.2 on page 51 provides a summary of the generic parameters supported by each layout algorithm. If you are using one of the subclasses provided with ILOG JViews, check the documentation for that subclass to know whether it supports a given parameter and whether it interprets the parameter in a particular way.

Allowed Time

Several layout algorithms can be designed to stop computation when a user-defined time specification is exceeded. This may be done for different reasons: as a security measure to avoid a long computation time on very large graphs or as an upper limit for algorithms that iteratively improve a current solution and have no other criteria to stop the computation.

ILOG JViews allows you to specify the allowed time:

```
void setAllowedTime(long time)
```

To obtain the current value, use the method:

```
long getAllowedTime()
```

If you subclass `IlvGraphLayout`, use the following method to know whether the specified time was exceeded:

```
boolean isLayoutTimeElapsed()
```

The time is in milliseconds. The default value is 32000 (32 seconds).

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, the following method is provided:

```
boolean supportsAllowedTime()
```

The default implementation returns `false`. A subclass can override this method to return `true` to indicate that this mechanism is supported.

Animation

Some iterative layout algorithms can optionally redraw the graph after each iteration or step. This may create a pleasant animation effect and may be used to keep the user aware of the evolution of the layout computation by showing intermediate results (as a kind of progress bar). However, this increases the duration of the layout because additional redrawing operations need to be performed.

ILOG JViews allows you to specify that a redraw of the grapher must be performed after each iteration (or step):

```
void setAnimate(boolean option)
```

To obtain the current value, use the following method:

```
boolean isAnimate()
```

The default value is `false`.

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, the following method is provided:

```
boolean supportsAnimation()
```

The default implementation returns `false`. A subclass can override this method to return `true` to indicate that this mechanism is supported.

Automatic Layout

For some layout algorithms, it may be suitable to have the layout automatically reperformed after each change of the graph, that is, when a node or link moves, is added, or is removed. Automatic layout is most useful for link layouts, in a situation, for example, where the shape of the links remain optimal after each editing action of the end-user. It also works well with other layout algorithms that offer an incremental behavior, that is, for which a small change of the graph usually produces only a small change of the layout. Automatic layout is generally not suitable for non-incremental layout algorithms.

To perform a layout automatically whenever the contents of a grapher change, use the following call:

```
layout.setAutoLayout(true);
```

Automatic layout works well if an `IlvGrapher` is not attached to other layouts. If multiple layouts are used for the same `IlvGrapher`, they may mutually affect each other. In this case, it is recommended to switch off automatic layout.

The following example shows how to perform multiple changes all at the same time in the `IlvGrapher` when automatic layout is switched on. An automatic layout is performed only once at the end of all the changes:

```
layout.attach(grapher);
layout.setAutoLayout(true);
   ...
// switch the notification of changes off
grapher.setContentsAdjusting(true);
try {
    // ... perform multiple changes without any automatic layout
    ...
} finally {
    // now the grapher notifies layout about the changes:
    // therefore, only one automatic layout is performed
    grapher.setContentsAdjusting(false);
}
```

For more information about automatic layout, see the method `performAutoLayout` in the *ILOG JViews Graph Layout Reference Manual*.

Coordinates Mode

The geometry, that is, the position and size, of the graphic objects that are used to represent nodes and links in an `IlvGrapher` is subject to a transformer (`IlvTransformer`). By default, the layout algorithms consider the geometry of the nodes and links of an `IlvGrapher` in a coordinate space that is appropriate for most cases. In some situations, it can be useful to specify a different coordinate space. For details, see *Choosing the Layout Coordinate Space* on page 323.

To specify the coordinate space, the following method is provided:

```
void setCoordinatesMode(int mode)
```

The valid values for `style` are:

- `IlvGraphLayout.MANAGER_COORDINATES`

  The geometry of the graph is computed using the coordinate space of the manager (that is, the attached `IlvGrapher`) without applying any transformation.

  This mode should be used if you visualize the graph at zoom level 1, or you do not visualize it at all, or the grapher contains only fully zoomable objects. In all these cases there is no need to take the transformer into account during the layout.

  Note that in this mode the dimensional parameters of the layout algorithms are considered as being specified in manager coordinates.

- `IlvGraphLayout.VIEW_COORDINATES`

  The geometry of the graph is computed in the coordinate space of the manager view. More exactly, all the coordinates are transformed using the current reference transformer.

  This mode should be used if you want the dimensional parameters of the layout algorithms to be considered as being specified in manager view coordinates.

- `IlvGraphLayout.INVERSE_VIEW_COORDINATES`

  The geometry of the graph is computed using the coordinate space of the manager view and then applying the inverse transformation. This mode is equivalent to the "manager coordinates" mode if the geometry of the graphic objects strictly obeys the transformer. (A small difference may exist because of the limited precision of the computations.)

  On the contrary, if some graphic objects are either nonzoomable or semizoomable (for example, links with a maximum line width), this mode gives different results than the manager coordinates mode. These results are optimal if the grapher is visualized using the same transformer as the one taken into account during the layout.

Note that in this mode the dimensional parameters of the layout algorithms are considered as being specified in manager coordinates.

The default mode is `IlvGraphLayout.INVERSE_VIEW_COORDINATES`.

To obtain the current choice, use the following method:

```
long getCoordinatesMode()
```

See also *Specifying the Coordinates Mode* on page 327.

Layout of Connected Components

The base class `IlvGraphLayout` provides generic support for the layout of a disconnected graph (composed of connected components). For details, see *Laying Out Connected Components of a Disconnected Graph* on page 304.

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, the following method is provided:

```
boolean supportsLayoutOfConnectedComponents()
```

Layout Region

Some layout algorithms can control the size of the graph drawing and can take into account a user-defined layout region. For example, for some layout algorithms you can specify that the layout should fit a given rectangle within a manager view or that it should fit the entire manager view.

ILOG JViews allows you to specify a layout region in three different ways:

- As the visible area of a given manager view (the argument `view`, usually the view that is used for displaying the grapher):

  ```
  void setLayoutRegion(IlvManagerView view)
  ```

- As a rectangle (the argument `rect`) with the dimensions being given in manager view coordinates (the argument `view`, usually the view that is used for displaying the grapher):

  ```
  void setLayoutRegion(IlvManagerView view, IlvRect rect)
  ```

- As a rectangle (the argument `rect`) with the dimensions being given in manager view coordinates:

  ```
  void setLayoutRegion(IlvRect rect)
  ```

You can use the last method if you want to perform the layout with no manager view attached to the grapher or if you want to define the layout region in manager coordinates.

You can obtain the current layout region setting using the following method:

```
IlvRect getSpecLayoutRegion()
```

This method returns a copy of the rectangle that defines the specified layout region. The dimensions of the rectangle are in the manager (grapher) coordinates. Depending on the last method you called, one of the following cases can occur:

- If `setLayoutRegion(IlvRect)` was the last method called, it returns a copy of the rectangle with no transformations.
- If `setLayoutRegion(IlvManagerView, IlvRect)` was the last method called, it returns a copy of the rectangle transformed to the manager coordinates using the transformer of the view. (The transformation to manager coordinates is not done if the coordinates mode is specified as *view coordinates*. For details, see *Coordinates Mode* on page 24)
- If `setLayoutRegion(IlvManagerView)` was the last method called, it returns a rectangle with the attributes `x=0, y=0` and with the attributes `width` and `height` equal to the current width and height of the view, transformed to manager coordinates using the current transformer of the view. (The transformation to manager coordinates is not done if the coordinates mode is specified as *view coordinates*.)
- None of the methods was called. (This is the default behavior.) If at least one manager view is attached to the grapher, it returns a rectangle with the attributes `x=0, y=0` and with the attributes `width` and `height` equal to the current width and height of the first attached view, transformed to manager coordinates using the transformer of the view. (The transformation to manager coordinates is not done if the coordinates mode is specified as *view coordinates*.) If no view is attached, the method returns `null`.

The layout algorithms call a different method:

```
IlvRect getCalcLayoutRegion()
```

This method first tries to use the layout region specification by calling the method `getSpecLayoutRegion()`. If this method returns a non-null rectangle, this rectangle is returned. Otherwise, the method tries to estimate an appropriate layout region according to the number and size of the nodes in the attached graph. If no graph is attached, or the attached graph is empty, it returns a default rectangle (`0`, `0`, `1000`, `1000`).

To indicate whether a subclass of `IlvGraphLayout` supports the layout region mechanism, the following method is provided:

```
boolean supportsLayoutRegion()
```

The default implementation returns `false`. A subclass can override this method in order to return `true` to indicate that this mechanism is supported.

> ***Note:*** *The implementation of the method* `void layout(boolean redraw)` *is solely responsible for whether the layout region is taken into account when calculating the layout, and in which manner. For details, refer to the documentation of the layout algorithms.*

Link Clipping

Some layout algorithms try to calculate the specific connection points of links at the border of nodes and require instances of `IlvRelativeLinkConnector` attached to the nodes, while other layout algorithms do not calculate any connection points but simply let the link connectors (any subclass of `IlvLinkConnector`) determine how the links connect to the nodes.

If a layout algorithm calculates specific connection points, then it places the connection points of links by default at the border of the bounding box of the nodes. If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want to place the connection points exactly on the border of the shape. This can be achieved by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape. Some examples are shown in Figure 2.1.

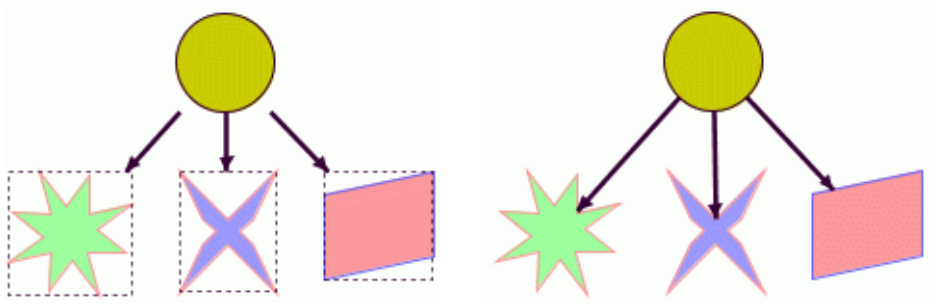


Figure 2.1 *Effect of Link Clipping Interface*

To indicate whether a subclass of `IlvGraphLayout` supports the link clip interface, the following method is provided:

```
boolean supportsLinkClipping()
```

To set a link clip interface, call:

```
void setLinkClipInterface(IlvLinkClipInterface interface)
```

You modify the position of the connection points of the links by implementing a class that implements the `IlvLinkClipInterface`. This interface defines the following method:

```
public IlvPoint getConnectionPoint(IlvGraphModel graphModel,
      Object node, IlvRect currentNodeBox, Object link,
      IlvPoint proposedConnectionPoint, IlvPoint auxControlPoint,
      boolean origin)
```

This method allows you to return the corrected connection point when the layout algorithm tries to connect to the proposed connection point. The `auxControlPoint` is the auxiliary control point of the link segment that ends at the proposed connection point. The flag `origin` indicates whether the connection point is the start point or the end point of the link,

One strategy is to calculate the intersection between the ray starting at `auxControlPoint` and going through `proposedConnectionPoint` and the shape of the node. If there is any intersection, we return the one closer to `auxControlPoint`. If there is no intersection, clipping is not possible and we return the proposed connection point. The following sample shows how to set a link clip interface that clips the connection points at the border of an ellipse or circle node:

```
layout.setLinkClipInterface(new IlvLinkClipInterface() {
    public IlvPoint getConnectionPoint(IlvGraphModel graphModel,
                                       Object node,
                                       IlvRect nodeBox,
                                       Object link,
                                       IlvPoint proposedConnectionPoint,
                                       IlvPoint auxControlPoint,
                                       boolean origin)
    {
      // get the intersections between the line through connect and control
      // point and the ellipse at currentNodeBox.
      IlvPoint[] intersectionPoints = new IlvPoint[2];
      int numIntersections = IlvGraphLayoutUtil.LineIntersectsEllipse(
                                 proposedConnectionPoint, auxControlPoint,
                                 nodeBox, intersectionPoints);
      // choose the result from the intersections
      return IlvGraphLayoutUtil.BestClipPointOnRay(proposedConnectionPoint,
                                                   auxControlPoint,
                                                   intersectionPoints,
                                                   numIntersections);
     }

});
```

> ***Note:*** *In addition to the link clip interface, the* `IlvClipLinkConnector` *can be used. This special link connector clips the links at nonrectangular node shapes and updates the connection points automatically during interactive node movements.*

Link Connection Box

If a layout algorithm calculates specific connection points, then it places the connection points of links by default at the border of the bounding box of the nodes, symmetrically with respect to the middle of each side. Sometimes it may be necessary to place the connection points on a rectangle smaller or larger than the bounding box, eventually in a nonsymmetric way. For instance, this can happen when labels are displayed below or above nodes (see Figure 2.2 on page 30). This can be achieved by specifying a link connection box interface. The link connection box interface allows you to specify, for each node, a node box different from the bounding box that is used to connect the links to the node.

To indicate whether a subclass of `IlvGraphLayout` supports the link connection box interface, the following method is provided:

```
boolean supportsLinkConnectionBox()
```

To set a link connection box interface, call:

```
void setLinkConnectionBoxInterface(IlvLinkConnectionBoxInterface
interface)
```

You implement the link connection box interface by defining a class that implements the `IlvLinkConnectionBoxInterface`. This interface defines the following method:

```
public IlvRect getBox(IlvGraphModel graphModel, Object node);
```

This method allows you to return the effective rectangle on which the connection points of the links are placed.

A second method defined on the interface allows the connection points to be “shifted” tangentially, in a different way for each side of each node:

```
public float getTangentialOffset(IlvGraphModel graphModel,
                                 Object node, int nodeSide);
```

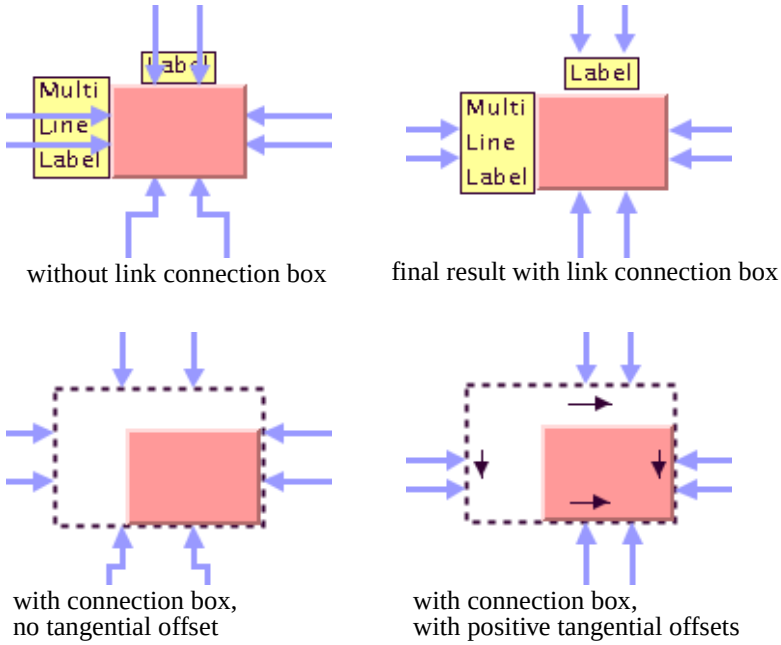


Figure 2.2 *Effect of Link Connection Box Interface*

For instance, to set a link connection box interface that returns a link connection rectangle that is smaller than the bounding box for all nodes of type `IlvShadowRectangle` and shifts up the connection points on the left and right side of all the nodes, call:

```
layout.setLinkConnectionBoxInterface(new IlvLinkConnectionBoxInterface() {
    public IlvRect getBox(IlvGraphModel graphModel, Object node) {
        IlvRect rect = graphModel.boundingBox(node);
        if (node instanceof IlvShadowRectangle) {
            // need a rect that is 4 pixels smaller
            rect.resize(rect.width-4.f, rect.height-4.f);
        }
        return rect;
    }
    public float getTangentialOffset(IlvGraphModel graphModel,
                                     Object node, int nodeSide) {
        switch (nodeSide) {
          IlvDirection.Left:
          IlvDirection.Right:
            return -10; // shift up with 10 for both left and right side
          IlvDirection.Top:
          IlvDirection.Bottom:
            return 0; // no shift for top and bottom side
    }
});
```

Some layout algorithms allow you to use the link connection box interface and the link clip interface in a combined way. It is specific to each layout algorithm how the interfaces will be used and which connection points are the final result.

Memory Savings

The computation of a layout on a large graph may require a large amount of memory. Some layout algorithms optionally use two ways to store data: one which gives the priority to speed (this is the default case), the other which consumes less memory and is usually slower. The amount of memory savings depends, of course, on the implementation of the subclass of `IlvGraphLayout`. No matter which option you choose for memory savings, the resulting layout should be the same.

To specify your option, use the following method:

```
void setMemorySavings(boolean option)
```

and to obtain the current value:

```
boolean isMemorySavings()
```

The default value is `false`.

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, the following method is provided:

```
boolean supportsMemorySavings()
```

The default implementation returns `false`. A subclass can override this method in order to return `true` to indicate that this mechanism is supported.

Percentage Completion Calculation

Some layout algorithms can provide an estimation of how much of the layout has been completed. This estimation is made available as a percentage value that is stored in the graph layout report. When the algorithm starts, the percentage value is set to 0. The layout algorithm calls the following method from time to time to increase the percentage value by steps until it reaches 100:

```
void increasePercentageComplete(int newPercentage);
```

The percentage value can be accessed from the layout report using the following:

```
int percentage = layoutReport.getPercentageComplete();
```

To see an example of how to read the percentage value during the running of a layout, see *Layout Event Listeners* on page 294.

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, the following method is provided:

```
boolean supportsPercentageComplete()
```

The default implementation returns `false`. A subclass can override this method to return `true` to indicate that this mechanism is supported.

Preserve Fixed Links

At times, you may want some links of the graph to be "pinned" (that is, to stay in their current shape when the layout is performed). You need a way to indicate the links that the layout algorithm cannot reshape. This makes sense especially when using a semiautomatic layout (the method where the end user fine tunes the layout by hand after the layout is completed) or when using an incremental layout (the method where the graph and/or the shape of the links is modified after the layout has been performed, and then the layout is performed again).

ILOG JViews allows you to specify that a link is fixed using the method:

```
void setFixed(Object link, boolean fixed)
```

If `fixed` is `true`, it means that the link is fixed. To obtain the current setting for a link:

```
boolean isFixed(Object link)
```

The default value is `false`.

To remove the fixed attribute from all links in the grapher, use the method:

```
void unfixAllLinks()
```

> ***Note:*** *The fixed attributes you may have set will be taken into consideration only if you call the method* `void setPreserveFixedLinks(boolean option)` *with a* `true` *argument.*

You can read the current option using the method:

```
boolean isPreserveFixedLinks()
```

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, the following method is provided:

```
boolean supportsPreserveFixedLinks()
```

The default implementation returns `false`. A subclass can override this method in order to return `true` to indicate that this mechanism is supported.

Preserve Fixed Nodes

At times, you may want some nodes of the graph to be "pinned" (that is, to stay in their current position when the layout is performed). You need a way to indicate the nodes that the layout algorithm cannot move. This makes sense especially when using a semiautomatic layout (the method where the end user fine tunes the layout by hand after the layout is completed) or when using an incremental layout (the method where the graph and/or the

position of the nodes is modified after the layout has been performed, and then the layout is performed again).

ILOG JViews allows you to specify that a node is fixed using the method:

```
void setFixed(Object node, boolean fixed)
```

If `fixed` is `true`, it means that the node is fixed. To obtain the current setting for a node:

```
boolean isFixed(Object node)
```

The default value is `false`.

To remove the fixed attribute from all nodes in the grapher, use the method:

```
void unfixAllNodes()
```

> ***Note:*** *The fixed attributes you may have set will be taken into consideration only if you call the method* `void setPreserveFixedNodes(boolean option)` *with a* `true` *argument.*

You can read the current option using the method:

```
boolean isPreserveFixedNodes()
```

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, the following method is provided:

```
boolean supportsPreserveFixedNodes()
```

The default implementation returns `false`. A subclass can override this method in order to return `true` to indicate that this mechanism is supported.

Random Generator Seed Value

Some layout algorithms use random numbers (or randomly chosen parameters) for which they accept a user-defined seed value. For example, the Random Layout uses the random generator to compute the coordinates of the nodes. The Spring Embedder uses the random generator to compute some internal variables.

Subclasses of `IlvGraphLayout` that are designed to support this mechanism allow the user to choose one of three ways of initializing the random generator:

- With a default value that is always the same.
- With a user-defined seed value that can be changed when re-performing the layout.
- With an arbitrary seed value, which is different each time. In this case, the random generator is initialized based on the system time.

The user chooses the initialization option depending on what happens when the layout algorithm is performed again on the same graph. If the same seed value is used, the same

layout is produced, which may be the desired result. In other situations, the user may want to produce different layouts in order to select the best one. This can be achieved by performing the layout several times using different seed values.

Here is an example of how this parameter can be used in combination with the `java.util.Random` class in your implementation of the method `IlvGraphLayout.layout()`:

```
Random random = (isUseSeedValueForRandomGenerator()) ?
  new Random(getSeedValueForRandomGenerator()) :
  new Random();
```

To specify the seed value, use the method:

```
void setSeedValueForRandomGenerator(long seed)
```

and to obtain the current value:

```
long getSeedValueForRandomGenerator()
```

The default value is `0`.

> ***Note:*** *The user-defined seed value is used only if you call the method* `void setUseSeedValueForRandomGenerator(boolean option)` *with a* `true` *argument.*

You can read the current option using the method:

```
boolean isUseSeedValueForRandomGenerator()
```

To indicate whether a subclass of `IlvGraphLayout` supports this parameter, the following method is provided:

```
boolean supportsRandomGenerator()
```

The default implementation returns `false`. A subclass can override this method in order to return `true` to indicate that this parameter is supported.

Save Parameters to Named Properties

The base class `IlvGraphLayout` provides support for saving the layout parameters (such as `isAnimate` or `isMemorySavings`) to `.ivl` files or to transfer the parameters to named properties. This is an advanced mechanism that is explained in detail in *Saving Layout Parameters and Preferred Layouts* on page 315.

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, the following method is provided:

```
boolean supportsSaveParametersToNamedProperties()
```

Stop Immediately

Several layout algorithms can stop computation when an external event occurs, for instance when the user hits a "Stop" button. To stop the layout, you can call:

```
boolean stopImmediately();
```

This method is typically called in a multithreaded application from a separate thread that is not the layout thread. The method returns `true` if the stop was initiated and `false` if the algorithm cannot stop. The method returns immediately, but the layout thread usually needs some additional time after initiating the stop to clean up data structures. The following code fragment illustrates the usage.

You start the layout in a separate thread:

```
Thread layoutThread = new Thread(new GraphLayoutPerformer(layout, grapher));
layoutThread.start();
```

The class `GraphLayoutPerformer` is an implementation of the interface `Runnable` that performs layout. The following is a sketch of this class:

```
class GraphLayoutPerformer implements Runnable
{
  ...
  public void run()
  {
    // from now we are busy
    busy = true;
    try {
      // perform the layout
      layout.performLayout(true, true);
    }
    catch (IlvGraphLayoutException e) {
      ... // handle the excepction
    }
    finally {
      // we are not busy anymore
      busy = false;
    }
  }
}
```

The Stop button operates outside the layout thread and simply calls the method `stopImmediately` of the running layout instance:

```
Button stopButton = new Button("Stop Layout");
stopButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (busy) layout.stopImmediately();
    }
});
```

> ***Note:*** *A detail has been omitted from the previous code fragment. A multitasking operation requires that the layout thread calls the* `yield()` *or* `sleep(t)` *methods from time to time. A good place to do this is by using a graph layout event listener. Event listeners are explained in Using Event Listeners on page 294.*

The consequences of stopping a layout process depends on the specific layout algorithm. Some layout algorithms have an iterative nature. Stopping the iteration process results in a slight loss of quality in the drawing, but the layout can still be considered valid. Other layout algorithms have a sequential nature. Interrupting the sequence of the layout steps may not result in a valid layout. Usually, these algorithms return to the situation before the start of the layout process.

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, the following method is provided:

```
boolean supportsStopImmediately()
```

Use Default Parameters

All the generic parameters have a default value. After modifying parameters, you may want the layout algorithm to use the default values. Then, you may want to return to your customized values. ILOG JViews keeps the previous settings when selecting the default values mode. You can switch between the default values mode and the mode for your own settings using the method:

```
void setUseDefaultParameters(boolean option)
```

To obtain the current value:

```
boolean isUseDefaultParameters()
```

The default value is `false`. This means that any setting you make will be taken into consideration and the parameters that have not been specified will have their default values.

C H A P T E R

3

Getting Started with Graph Layout

This chapter provides information to get started using the Graph Layout module of ILOG JViews. The following topics are covered:

- *Basic Steps for Using Layout Algorithms: A Summary*
- *Sample Java Application*

Basic Steps for Using Layout Algorithms: A Summary

To use the layout algorithms provided by the Graph Layout module of ILOG JViews, you will usually perform the following steps:

1. Create a grapher object (`IlvGrapher`) and fill it with nodes and links.
2. Create an instance of the layout algorithm (any subclass of `ilog.views.graphlayout.IlvGraphLayout`).
3. Declare a handle for the corresponding layout report. The layout report is an object in which the layout algorithm stores information about its behavior. For details see *Using a Layout Report* on page 292.
4. Attach the grapher to the layout instance.
5. Modify the default settings for the layout parameters, if needed.
6. Call the performLayout method inside a `try` block.
7. Read and display information from the layout report.
8. Catch the exceptions.
9. When the layout instance is no longer needed, detach the grapher from the layout instance.

An application that illustrates these steps is provided in *Sample Java Application* on page 38.

Sample Java Application

You can use this application as an example to get started with the layout algorithms of the Graph Layout module. The example uses the Orthogonal Link Layout, but the principles are mostly the same for any of the other layouts.

The source code of the application is named `LayoutSample1.java` and can be found at the location:

`<installdir>/doc/usermansrc/graphlayout/LayoutSample1.java`

To compile and run the example, do the following:

1. Go to the `graphlayout` directory at the above path. (On Windows 98-2000-ME-NT-XP, you must open a DOS Console).

2. Set the CLASSPATH variable to the ILOG JViews library and the current directory.

   On Windows 98-2000-NT-XP: `.;<installdir>\classes\jviewsall.jar`

   On Unix: `.:<installdir>/classes/jviewsall.jar`

3. Compile the application:

   ```
   javac LayoutSample1.java
   ```

4. Run the application:

   ```
   java LayoutSample1
   ```

The `LayoutSample1.java` contains the following code:

```
// the ILOG JViews Graphic Framework
import ilog.views.*;
// the ILOG JViews Graph Layout Framework
import ilog.views.graphlayout.*;
// the ILOG JViews Link Layout
import ilog.views.graphlayout.link.*;
// the Java AWT package
import java.awt.*;

public class LayoutSample1
{
  public static final void main(String[] arg)
  {
    // Create the layout instance
    IlvLinkLayout layout = new IlvLinkLayout();
    // Create the grapher instance
    IlvGrapher grapher = new IlvGrapher();
    // Create the manager view instance
    IlvManagerView view = new IlvManagerView(grapher);
     // An AWT Frame to display
    Frame frame = new Frame("Layout Sample");
    // The name of the IVL file containing the graph data
    final String fileName = "SampleGraph1.ivl";

    // Put the manager view inside an AWT Frame and show it
    frame.add(view);
    frame.setSize(600, 600);
    frame.setVisible(true);

    // Fill the grapher with nodes and links from a JViews IVL file.
    // Alternatively, the nodes and links could be created by code.
    try {
      grapher.read(fileName);
    } catch (Exception e) {
      System.out.println("could not read " + fileName);
      return;
    }
```

```
    // Attach the grapher to the layout instance
    layout.attach(grapher);
    try {
      // Perform the layout and get the layout report
      IlvGraphLayoutReport layoutReport = layout.performLayout();

      int code = layoutReport.getCode();

      // Print information from the layout report (optional)
      System.out.println("layout done in " +
                         layoutReport.getLayoutTime() +
                         " millisec., code = " +
                         code + " (" +
                         layoutReport.codeToString(code) + ")");
    }

    // Catch the exceptions
    catch (IlvGraphLayoutException e) {
      System.out.println(e.getMessage());
    }

    // Detach the grapher from the layout instance
    layout.detach();
  }
  // ... further methods ...
}
```

The sample Java application produces the graph shown in Figure 3.1:

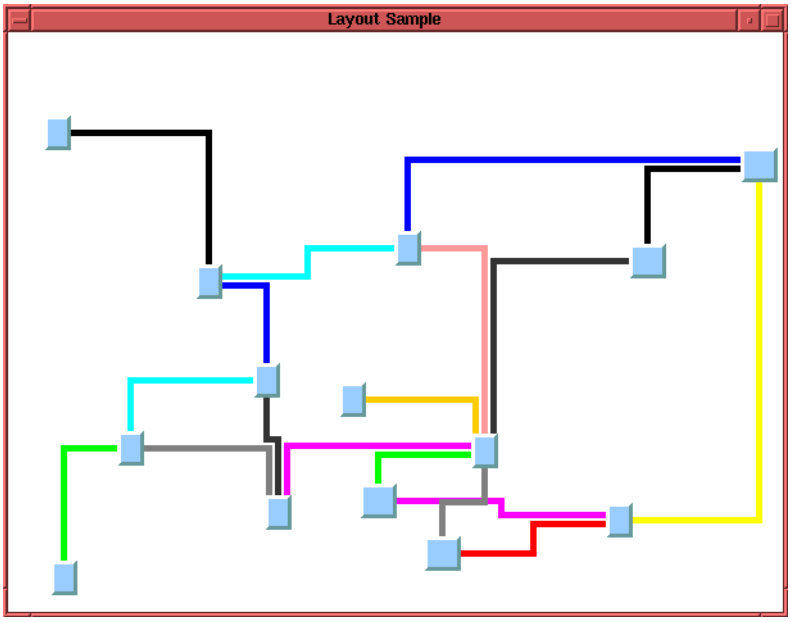


Figure 3.1 *Output from Sample Java Application*

C H A P T E R

4

Layout Algorithms

This chapter describes the layout algorithms of the ILOG JViews Graph Layout module. The following topics are covered:

- *Determining the Appropriate Layout Algorithm*
- *Topological Mesh Layout (TML)*
- *Spring Embedder Layout*
- *Uniform Length Edges Layout*
- *Tree Layout*
- *Hierarchical Layout*
- *Link Layout*
- *Random Layout*
- *Bus Layout*
- *Circular Layout*
- *Grid Layout*
- *Recursive Layout*
- *Multiple Layout*

Determining the Appropriate Layout Algorithm

When using the Graph Layout module, you need to determine which of the ready-to-use layout algorithms is appropriate for your particular needs. Some layout algorithms can handle a wide range of graphs. Others are designed for particular classes of graphs and will give poor results or will reject graphs that do not belong to these classes. For example, a Tree Layout algorithm is designed for tree graphs, but not cyclic graphs. Therefore, it is important to lay out a graph using the appropriate layout algorithm.

Table 4.1 can help you determine which of the layout algorithms is best suited for a particular type of graph.

- Across the top of the table are various classifications of different types of graphs.
- The layout algorithms appear on the left side of the table.
- Table cells containing illustrations indicate when a layout algorithm is applicable for a particular type of graph.

By identifying the general characteristics of the graph you want to lay out, you can see from the table whether a layout algorithm is suited for that particular type of graph.

For example, if you know that the structure of the graph is a tree, you can look at the Domain-Independent Graphs/Trees column to see which layout algorithms are appropriate. The Spring Embedder Layout, Uniform Length Edges Layout, Tree Layout, and Hierarchical Layout could all be used. Use the illustrations in the table cells to help you further narrow your choice.

The Recursive Layout can be used to control the layout of nested graphs (containing subgraphs and intergraph links). This is in particular useful if different layout styles should be applied to different subgraphs. The other layout algorithms such as Spring Embedder Layout, Tree Layout, and Hierarchical Layout treat only flat graphs (unless otherwise noted), that is, a specific layout instance is only able to lay out the nodes and links of the attached graph, but not the nodes and links of its subgraphs. The Recursive Layout allows you to specify which flat layout is used for which subgraph, and it traverses the entire nested graph recursively when applying the layout. As result, the entire nested graph is laid out.

The Multiple Layout can be used to combine several different layouts into one instance. In this case, they become *sublayouts* of the Multiple Layout instance. This is useful in particular for nested graphs when used in combination with the Recursive Layout. The Multiple Layout ensures that the normal layout, the routing of the intergraph links, and the layout of labels are applied in the correct order to a nested graph.

Table 4.1 *Layout Algorithms and Common Types of Graphs*

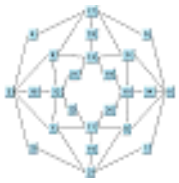
Layout	Domain-Independent Graphs				Telecom-Oriented Representations
	Trees	**Cyclic Graphs**	**Combination of Cycles and Trees**	**Any Graph**	
Topological Mesh Layout			Requires (semi)manual refinements		
Spring Embedder Layout		Preferable to avoid heavily interconnected graphs (large number of cycles)			

Table 4.1 *Layout Algorithms and Common Types of Graphs (Continued)*

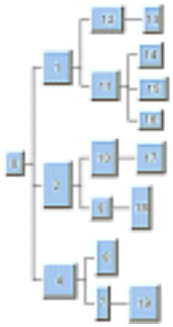
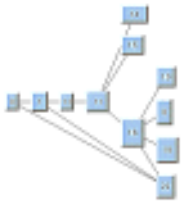
Layout	Domain-Independent Graphs				Telecom-Oriented Representations
	Trees	**Cyclic Graphs**	**Combination of Cycles and Trees**	**Any Graph**	
Uniform Length Edges Layout		Preferable to avoid heavily interconnected graphs (large number of cycles)			
Tree Layout					

Table 4.1 *Layout Algorithms and Common Types of Graphs (Continued)*

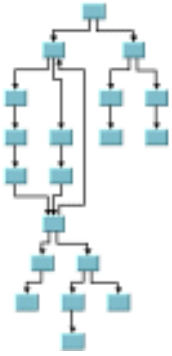
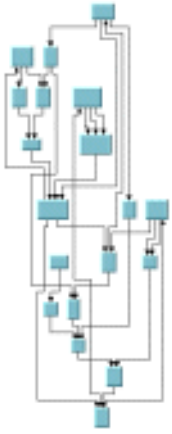
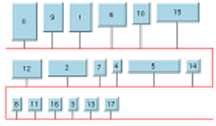
Layout	Domain-Independent Graphs				Telecom-Oriented Representations
	Trees	Cyclic Graphs	Combination of Cycles and Trees	Any Graph	
Hierarchical Layout					
Link Layout					
Bus Layout					 For bus topologies

Table 4.1 *Layout Algorithms and Common Types of Graphs (Continued)*

Layout	Domain-Independent Graphs				Telecom-Oriented Representations
	Trees	Cyclic Graphs	Combination of Cycles and Trees	Any Graph	
Circular Layout					For interconnected ring/star topologies
Grid Layout				Note that the algorithm does not take into account the links between the nodes.	
Recursive Layout				Nested graphs.	
Multiple Layout				Combination of multiple different layout algorithms on the same graph (in particular for nested graphs).	

Typical Ways for Choosing the Layout

Determining an appropriate algorithm for the graph can be done either by the end user at run time or by the programmer of the application. This process can be semiautomatic, where user intervention is involved, or automatic, where the application does everything with no user intervention.

- Semiautomatic layout

  For applications using a semiautomatic layout, the choice of the layout algorithm is done by the end user. The application can provide a menu or some other way to select the layout algorithm.

  In some cases, this may be an iterative process. The user may try different layout algorithms with different values for the parameters and/or may apply manual refinements in order to find the best layout. Eventually, the application can provide some help using textual explanations or by automatically checking the graph to find out to which class it belongs. For example, to detect whether the graph that has been attached to a layout instance is a tree, the `IlvGraphLayoutUtil` class provides the following method:

  ```
  static boolean IsTree(IlvGraphLayout layout, Object startNode)
  ```

  For details on this method, see the corresponding section of the reference manual.

- Automatic layout

  If an automatic layout is needed, the choice of the layout algorithm can be:

  - Dynamically chosen at run time using heuristics or rules to determine the appropriate layout algorithm depending on the structure and/or size of the graph.
  - Hard-coded if the developer knows what types of graphs will be used and can determine the appropriate layout algorithm.

Procedure for Dynamically Choosing the Layout Algorithm

If nothing is known about the graphs that the application will need to lay out, the developer can write a routine that automatically chooses the layout algorithm at run-time. The following simple rules could be applied:

1. If the nodes of the graph cannot be moved (they are geo-positioned), use the Link Layout.
2. If the graph is a tree, use the Tree Layout.
3. Otherwise, use one of the layout algorithms that are the less restricted to a given graph category, especially the Uniform Length Edges Layout. (The preferred length of the links could also be computed with respect to the size of the nodes.)

4. If the graph is too large, apply a "divide-and-conquer" strategy. Cut the graph into several subgraphs and apply the layout separately to each subgraph. If the graph is disconnected, you can use the built-in support provided by the layout library to perform this task automatically. (See *Layout of Connected Components* on page 25.)
5. If the graph is nested, use the Recursive Layout algorithm that controls which subgraph is laid out by which (flat) sublayout. Use step 1-4) above to determine the sublayouts for the subgraphs.

Procedure for Hard-coding when Layout is Applied at Programming Time

A special case occurs when the application will deal with only a small set of graphs that are known at the time the application is built. In this case, the layout can be performed at programming time. A possible step-by-step procedure may be the following:

1. Create each graph manually with a graph editor or by code.
2. Try different layout algorithms and choose the best for each graph.
3. Apply manual refinements to the layout if needed.
4. Store the result of the layout by saving the graphers in `.ivl` files.
5. Provide these files with the application.
6. When the application is used, these files will simply be loaded. (There will be no need to perform the layout again since it is already done.)

Procedure for Hard-coding when Layout is Applied at Run Time

If the choice of the layout algorithm is hard-coded, but the layout must be performed at run time because the graphs are not known at programming time, one possible step-by-step procedure for the choice of the appropriate layout algorithm may be the following:

1. Look at sample graphs for your domain.
2. Try to determine some generalities about the properties of the structure and the size of the graph (Is the graph cyclic? Is the graph a tree? Is the graph a combination of the two? What is the number of nodes and links in the graph?)
3. Pick one of the corresponding layout algorithms from Table 4.1 on page 45.
4. Try out the algorithm on one or more samples.

Generic Features and Parameters Support

The generic features and parameters of the Graph Layout module allow you to customize the behavior of the layout algorithms to meet specific needs and to perform useful operations such as saving the layout parameters in a file. Table 4.2 indicates the generic features and parameters that are supported by each layout algorithm. These parameters are defined in the base class for all layout algorithms, `IlvGraphLayout`.

Table 4.2 *Generic Parameters Supported by Layout Algorithms*

Layout Algorithm	Allowed Time	Animation	Fixed Links	Fixed Nodes	Layout of Connected Components	Layout Region	Link Clipping	Link Connection Box	Memory Savings	Percentage Complete	Random Generator Seed Value	Save Parameters to File	Stop Immediately
Topological Mesh Layout (TML)	Yes	Yes		Yes	Yes	Yes	Yes	Yes	Yes			Yes	Yes
Spring Embedder Layout	Yes	Yes		Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Uniform Length Edges Layout	Yes	Yes		Yes	Yes	Yes	Yes	Yes				Yes	Yes
Tree Layout	Yes		Yes	Yes	Yes		Yes	Yes		Yes		Yes	Yes
Hierarchical Layout	Yes		Yes	Yes	Yes		Yes	Yes		Yes		Yes	Yes
Link Layout	Yes	Yes	Yes					Yes				Yes	Yes
Random Layout	Yes			Yes		Yes				Yes	Yes	Yes	Yes
Bus Layout	Yes			Yes	Yes	Yes	Yes	Yes				Yes	Yes
Circular Layout				Yes	Yes	Yes	Yes	Yes				Yes	
Grid Layout	Yes			Yes		Yes						Yes	Yes
Recursive Layout	Yes									Yes		Yes	Yes
Multiple Layout	Yes				Yes					Yes		Yes	Yes

Layout Characteristics

It is often useful to know how certain settings will affect the resulting layout of the graph after the layout algorithm has been applied. Table 4.3 provides additional information about the behavior of the layout algorithms.

Table 4.3 *Layout Characteristics of Layout Algorithms*

Layout Algorithm	Do the initial positions of the nodes affect the layout?[1]	How do I get a different layout of the same graph when I perform the layout a second time?
Topological Mesh Layout (TML)	No	You can completely change the layout by using the starting node, outer cycle, and fixed nodes parameters. To change only the dimensions of the graph, use the layout region parameter.
Spring Embedder Layout	No	This is the default behavior when using the default parameter settings (the random generator is initialized differently each time). To change only the dimensions of the graph, use the layout region and spring constant parameters.
Uniform Length Edges Layout	Yes	You can completely change the layout by changing the initial positions of the nodes. To change only the dimensions of the graph, use the preferred length of the links or size of the layout region.
Tree Layout	Yes (if incremental mode is switched on)	In incremental mode, you can change the layout by changing the initial positions of the nodes. Furthermore, you can change the layout by selecting a different root node. To change only the dimensions of the graph, use the various offset parameters.
Hierarchical Layout	Yes (if incremental mode is switched on)	In incremental mode, you can change the layout by changing the initial positions of the nodes. Furthermore, you can use specified node level indices to change the level structure. You can use specified node position indices to change the node order within the levels. You can change the layout by changing the link priorities. To change only the dimensions of the graph, use the various offset parameters.

Table 4.3 *Layout Characteristics of Layout Algorithms (Continued)*

Layout Algorithm	Do the initial positions of the nodes affect the layout?[1]	How do I get a different layout of the same graph when I perform the layout a second time?
Link Layout	Yes	Link Layout routes the links depending on the node positions. It does not move the nodes. You can change the link style option and the dimensional parameters, such as the link offset and final segment length. You can also specify the rules for computing the connection points of the links.
Random Layout	No	This is the default behavior when using the default parameter settings (the random generator is initialized differently each time).
Bus Layout	No, except in incremental mode	You change the dimensions of the graph by using the various dimensional parameters.
Circular Layout	No	You can completely change the layout by using clustering settings and the root clusters parameter. You can change the dimensions of the graph by using the dimensional parameters.
Grid Layout	No, except in incremental mode	You can change various dimensional parameters, layout mode, and so on.
Recursive Layout	Depends on the behavior of the sublayouts applied to the subgraphs.	Depends on the behavior of the sublayouts applied to the subgraphs. You can change the parameters of the sublayouts individually.
Multiple Layout	Depends on the behavior of the sublayout that is applied first.	Depends on the behavior of the sublayouts of the Multiple Layout instance. You can change the parameters of the sublayouts individually.

[1] All of the layout classes provided in ILOG JViews (except the Link Layout) support the fixed nodes mechanism. This means that you can specify nodes that cannot be moved during the layout.

Topological Mesh Layout (TML)

In this section, you will learn about the *Topological Mesh Layout (TML)* algorithm from the ILOG JViews Graph Layout module (class `IlvTopologicalMeshLayout` from the package `ilog.views.graphlayout.topologicalmesh`).

Samples

Here are some sample drawings produced with TML:

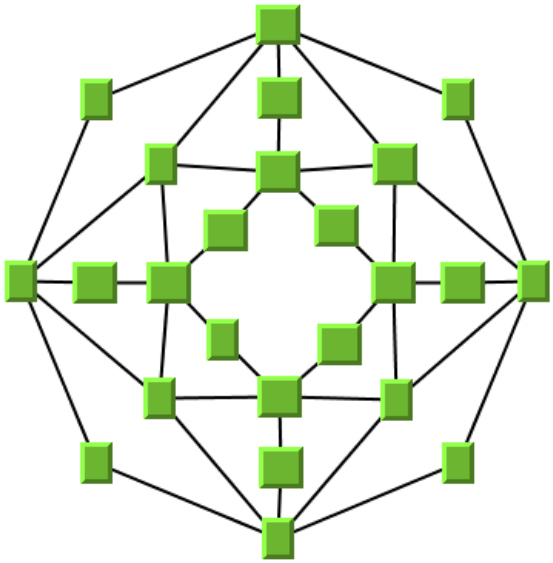


Figure 4.1 *Small Cyclic Graph Drawing Produced with TML*

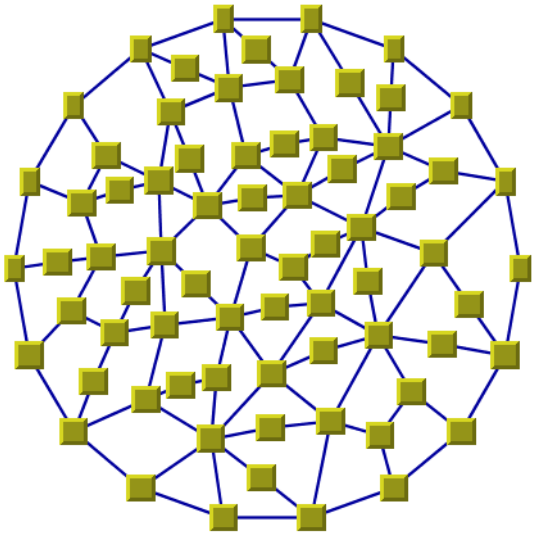


Figure 4.2 *Large Cyclic Graph Drawing Produced with TML*

What Types of Graphs?

- Cyclic (2-connected) graphs. (Preferably without cut-nodes or cut-edges; otherwise, manual adjustments are necessary.)
- Cyclic (2-connected) graphs plus only a few branches. (You may need to make manual adjustments for the branches.)
- Both planar and nonplanar graphs.

Application Domains

Application domains of the Topological Mesh Layout include:

- Database and knowledge engineering (semantic networks, qualitative reasoning and other artificial intelligence diagrams)

Features

- Most of the time, produces planar drawings of planar graphs, and drawings with a small number of link crossings for nonplanar graphs.
- Produces a nice layout for most small- and medium-size graphs relatively quickly. (The maximum cyclomatic number of the graph is about 30-50, but the number of nodes and links can be a lot higher.)

- Most of the time, produces symmetrical drawings of symmetrical graphs.
- The computation time for one iteration depends on the cyclomatic number of the graph, which is smaller than the number of nodes or links.
- The user can obtain several layouts of the same graph easily and quickly by simply changing a parameter (especially the starting node and the outer cycle) or by applying manual refinements to the layout. The best layout can then be selected from the resulting layouts.

Limitations

- The algorithm tries to minimize the number of link crossings (which is generally an NP-complete problem). It is mathematically impossible to quickly solve this problem for any graph size. Therefore, the algorithm uses heuristics that cannot always obtain a layout with the theoretical minimum number of link crossings.
- The computation time required to obtain an appropriate drawing grows relatively quickly with the cyclomatic number and the layout process may become very time-consuming for large graphs. Again, this is because the minimization of the number of link crossings is mathematically NP-complete in the general case.
- The algorithm cannot automatically produce appropriate drawings of some types of graphs:
  - For graphs containing branches and graphs containing cut-nodes or cut-edges, manual adjustments are necessary. (See *Refining a Graph Layout* on page 66.)
  - For disconnected graphs, the connected component layout feature should be used. (See *Layout of Connected Components* on page 25.)
- The layout algorithm often produces a drawing with no overlapping nodes. Nevertheless, overlapping nodes cannot always be avoided. When overlapping occurs, you can try to increase the size of the layout region parameter or to change the outer cycle (see the method `setExteriorCycleId`). You can also use manual adjustments to correct the problem.

Brief Description of the Algorithm

TML is an heuristical approach for the layout of cyclic (2-connected) graphs, either planar or nonplanar graphs.

TML is basically very simple to use. However, to use all the functionalities of TML, you should understand its basic concepts.

TML Basic Concepts

When laying out a general graph, producing a drawing with a minimum number of link crossings is a mathematically NP-complete problem. The search space (and time) is exponentially combinatorial. Traditionally, most of the existing layout algorithms use node coordinates from the beginning, searching for a coordinate set to minimize the cost function, which is mainly the number of link crossings. These coordinates can be constrained on a grid, but the number of combinations to explore is still enormous.

In contrast, TML uses a **two-step approach** that drastically reduces the number of combinations to explore. The first step of TML deals only with the pure topology (that is, the connectivity) of the graph without taking into consideration the node coordinates. This first step is called **topological optimization.** It chooses one of the cycles of the graph to be used in the second step.

In the second step, called **nodes placement**, the result of the first step is used to compute the coordinates of the nodes using a deterministic, high-speed barycenter algorithm. Of course, the problem still remains NP-complete and large graphs cannot be processed. In practice, however, you will often get better results for "mesh" graphs with TML than with many other algorithms.

Step 1: Topological Optimization

Input: The topology of the graph (its connectivity or the neighborhood relationships between nodes).

Output: A set of possible outer cycles, ordered decreasingly by their lengths. The length of a cycle is the number of nodes in the cycle.

Explanation: This step determines what cycles of the graph, if used as an outer cycle for drawing the graph in Step 2, will allow a drawing with a minimum number of link crossings. An optimization algorithm tries to minimize a heuristic cost function that estimates the number of link crossings for each solution, based on pure topology (graph connectivity).

Step 2: Nodes Placement

Input: The output of Step 1 and the graph.

Output: A set of coordinates for the nodes. The coordinates are assigned to the nodes to obtain the graph drawing.

Explanation: This step is a variant of the "barycentric" layout algorithm. It takes a cycle from the output of Step 1 and draws it as a regular polygon. Then, it iteratively moves each node (except those on the regular polygon) at the "barycenter" of its neighbors (the nodes to which it is connected). This procedure always converges, and the final result is a graph drawing where the number of link crossings is dependent only on the choice of the outer cycle.

Code Sample

Below is a code sample using the `IlvTopologicalMeshLayout` class:

```
 ...
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.topologicalmesh.*;
 ...
IlvGrapher grapher = new IlvGrapher();
IlvManagerView view = new IlvManagerView(grapher);

 ... /*   Fill in the grapher with nodes and links here */

IlvTopologicalMeshLayout layout = new IlvTopologicalMeshLayout();
layout.attach(grapher);
try {
        IlvTopologicalMeshLayoutReport layoutReport =
              (IlvTopologicalMeshLayoutReport)layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Generic Features and Parameters

TML supports the following generic parameters defined in the `IlvGraphLayout` class:

- *Allowed Time*
- *Animation*
- *Layout of Connected Components*
- *Layout Region*
- *Link Clipping*
- *Link Connection Box*
- *Memory Savings*
- *Preserve Fixed Links*
- *Preserve Fixed Nodes*
- *Save Parameters to Named Properties*
- *Stop Immediately*

The following paragraphs describe the particular way in which these parameters are used by this subclass.

Note that all the methods allowing the modification of these parameters are overridden in this subclass. This class keeps track of the changes for parameters that may affect the result of Step 1 separately from the parameters that may affect only Step 2. In this way, Step 1 is not repeated. The previous results are used if no parameters were modified since the last time the layout was successfully performed on the same graph using the same layout instance.

Allowed Time

Step 1 of TML stops if the allowed time setting has elapsed. In the same manner, Step 2 of TML stops if the allowed time is exceeded. (See *Allowed Time* on page 22.)

You can specify separate time settings for each step. Each step is stopped if its specified time limit is exceeded. To learn how to do this, see *Optimization Iterations and Allowed Time* on page 62 and *Nodes Placement Iterations and Allowed Time* on page 62.

Animation

In TML, the animation parameter (see *Animation* on page 22) is used only in Step 2. If specified, a redraw will be automatically performed after each iteration of the step. This is useful to better understand what is happening during the step and especially if you want to be able to choose the fixed nodes in a manual refinement procedure. (See *Using Fixed Nodes* on page 66.)

Layout of Connected Components

The layout algorithm can use the generic mechanism to lay out connected components. (For more information about this mechanism, see *Layout of Connected Components* on page 25.)

If the generic connected component layout mechanism is disabled, the algorithm lays out only the connected component that contains the starting node.

Layout Region

Step 2 of TML first draws the outer cycle computed in Step 1 as a regular polygon. It uses the layout region setting (either your own or the default setting) to choose the size and the position of the polygon. The remaining nodes are moved inside this polygon. All three ways to specify the layout region are available for this subclass. (See *Layout Region* on page 25.)

Remember that if you are using the default settings, the visible area of the manager view (an instance of `IlvManagerView`) attached to the grapher is used as a layout region. If several manager views are attached, the first attached view is used. If no manager view is attached, the layout region is automatically estimated on the basis of the number and size of the nodes.

If TML produces a layout with overlapping nodes, one possible way to correct the problem is to increase the size of the layout region. (For details, see *Using the Layout Region Parameter* on page 68.)

Link Clipping

The layout algorithm can use a link clip interface to clip the end points of a link. (See *Link Clipping* on page 27.)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip interface is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using a Link Clipping Interface* on page 69 for details of the link clipping mechanism in TML.

Link Connection Box

The layout algorithm can use a link connection box interface (see *Link Connection Box* on page 29) in combination with the link clip interface. If no link clip interface is used, the link connection box interface has no effect. For details see *Using a Link Clipping Interface* on page 69.

Memory Savings

As with all classes supporting this parameter, a certain amount of memory savings can be obtained by selecting this option. Note that using this option does not change the resulting layout. It just slows down the computation. (See *Memory Savings* on page 31.)

Preserve Fixed Links

TML does not reshape the links that are specified as fixed. (See *Preserve Fixed Links* on page 32. See also *Link Style* on page 61.)

Preserve Fixed Nodes

TML does not move the nodes that are specified as fixed. Moreover, the algorithm takes into account the fixed nodes when computing the position of the nonfixed nodes. (See *Preserve Fixed Nodes* on page 32.)

If TML produces a layout with overlapping nodes, you can use the fixed nodes mechanism to correct the problem. (For details, see *Using Fixed Nodes* on page 66.)

Save Parameters to Named Properties

The layout algorithm can save its layout parameters into named properties. This can be used to save layout parameters to `.ilv` files. (For a detailed description of this feature, see *Save Parameters to Named Properties* on page 34 and *Saving Layout Parameters and Preferred Layouts* on page 315).

Stop Immediately

The layout algorithm stops after cleanup if the method `stopImmediately` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop Immediately* on page 35.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific Parameters

The following parameters are specific to the `IlvTopologicalMeshLayout` class.

Link Style

When the layout algorithm moves the nodes, straight-line links (such as instances of `IlvLinkImage`) will automatically "follow" the new positions of their end nodes. If the grapher contains other types of links (for example, `IlvPolylineLinkImage` or `IlvSplineLinkImage`), the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can ask the layout algorithm to automatically remove all the intermediate points of the links (if any).

To do this, the following method is provided:

```
void setLinkStyle(int style)
```

The valid values for `style` are:

- `IlvTopologicalMeshLayout.NO_RESHAPE_STYLE`

  None of the links is reshaped in any manner.

- `IlvTopologicalMeshLayout.STRAIGHT_LINE_STYLE`

  All the intermediate points of the links (except for links specified as fixed) are removed. This is the default value.

> ***Note:*** *If* `STRAIGHT_LINE_STYLE` *is selected, the layout algorithm may raise an* `IlvInappropriateLinkException` *when the layout is performed on an* `IlvGrapher`*, and links that cannot be given a straight-line shape are used. Such links are* `IlvOneLinkImage`, `IlvDoubleLinkImage`, `IlvOneSplineLinkImage`, *and* `IlvDoubleSplineLinkImage`*. You can use the methods* `EnsureAppropriateLinkTypes` *or* `EnsureAppropriateLinks` *defined in* `IlvGraphLayoutUtil` *to automatically replace inappropriate links. This can be done before layout or when the* `IlvInappropriateLinkException` *is caught. For details on these methods, see the ILOG JViews Graph Layout Reference Manual. For details on the graph model, see Using the Graph Model on page 297.*

To obtain the current choice, use the following method:

```
long getLinkStyle()
```

Optimization Iterations and Allowed Time

The iterative computation performed in Step 1 is stopped if the number of iterations exceeds the allowed number of iterations for optimization or the time exceeds the allowed time for optimization (or, of course, if the general layout time has elapsed).

To specify the parameters, use the following methods:

```
void setAllowedOptimizationTime(long time)
```

```
void setAllowedNumberOfOptimizationIterations(int iter)
```

The `time` is in milliseconds. The default value is 28000 (28 seconds).

To obtain the current values, use the following methods:

```
long getAllowedOptimizationTime()
```

```
int getAllowedNumberOfOptimizationIterations()
```

Nodes Placement Iterations and Allowed Time

The iterative computation performed in Step 2 is stopped if the number of iterations exceeds the allowed number of iterations or the time exceeds the allowed time for nodes placement (or, of course, if the general layout time has elapsed).

ILOG JViews allows you to specify these parameters using the methods:

```
void setAllowedNodesPlacementTime(long time)
```

```
void setAllowedNumberOfNodesPlacementIterations(int iter)
```

The `time` is in milliseconds. The default value is 28000 (28 seconds).

To obtain the current values, use the following methods:

```
long getAllowedNodesPlacementTime()
```

```
int getAllowedNumberOfNodesPlacementIterations()
```

Nodes Placement Algorithm

Two “barycentric” algorithms are implemented for Step 2 of TML. You can specify the algorithm using the method:

```
void setNodesPlacementAlgorithm(int option)
```

The valid values for `option` are:

- `IlvTopologicalMeshLayout.SLOW_GOOD`

  This option provides more uniformity of the nodes distribution inside the outer cycle, but is slightly slower.

- `IlvTopologicalMeshLayout.QUICK_BAD`

  This option provides less uniformity of the nodes distribution, but is slightly quicker.

To obtain the current choice, use the method:

```
int getNodesPlacementAlgorithm()
```

In most cases, both algorithms are fairly quick. We recommend that you use the `SLOW_GOOD` version, which is the default value. Compare the layouts of the same graph in Figure 4.3 and Figure 4.4 to get an idea of the difference between these algorithms.

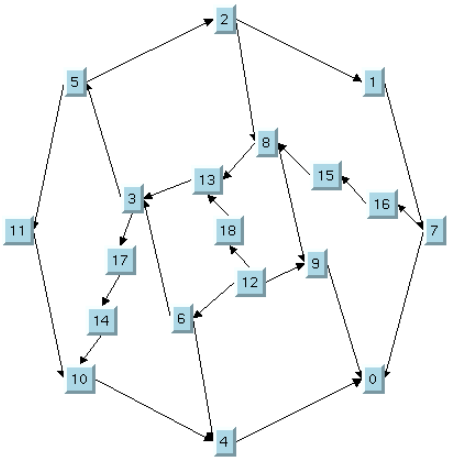


Figure 4.3 *Nodes Placement Algorithm:* `SLOW_GOOD`

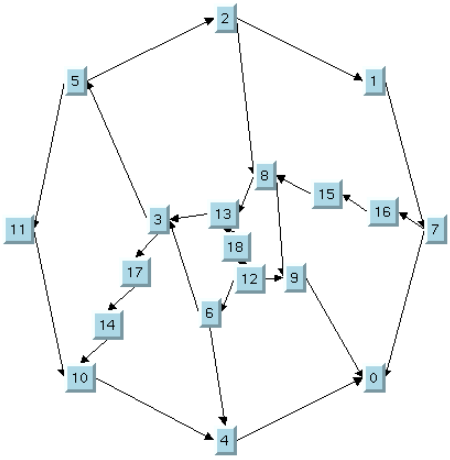


Figure 4.4 *Nodes Placement Algorithm:* `QUICK_BAD`

Starting Node

Step 1 of TML needs a "starting node" parameter. TML produces a different layout depending on the choice of the starting node. (TML works on different cycles of the graph depending on the starting node.)

To specify the starting node, use the method:

```
void setStartingNode(Object node)
```

To obtain the current choice for the starting node, use the method:

```
Object getStartingNode()
```

If you do not specify a starting node, an arbitrary node of the graph is used.

If you specify a starting node, but want to reset the setting and return to the default behavior (that is, having an arbitrary node automatically chosen as the starting node), call the `setStartingNode` method with a `null` argument.

Outer Cycle

Step 1 of TML computes a set of cycles that can be used as the outer cycle in Step 2. By default, the longest cycle is actually used (that is, the cycle containing the largest number of nodes). However, you may find it useful to try a different outer cycle. In this case, you can use the method:

```
void setExteriorCycleId(int cycleId)
```

The valid values for `cycleId` range from zero to the number of cycles computed by Step 1 minus one. This number is returned by the method:

```
int getNumberOfPossibleExteriorCycles()
```

If the number is not in this range, the value zero is used. To obtain the current choice for the starting node, use the method:

```
int getExteriorCycleId()
```

You can use these methods only after having successfully performed the layout. Otherwise, no outer cycle is defined.

When the layout is performed again with a new outer cycle, only Step 2 of TML is performed, and not the time-consuming Step 1. This is true if the topology of the graph has not been changed (that is, no nodes or links were added or removed), and no parameters that affect Step 1 have been changed.

The following drawings show various layouts produced **for the same graph** when the `cycleId` parameter is changed:

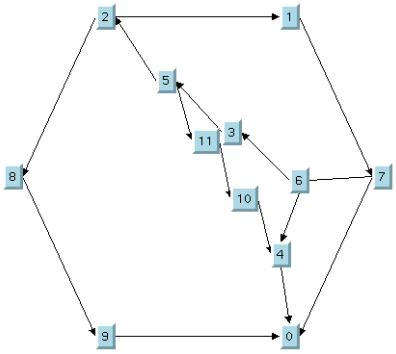


Figure 4.5 *Layout Using 1st Outer Cycle*

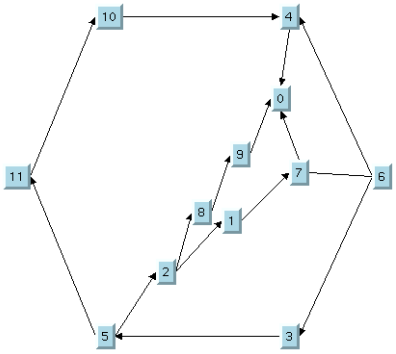


Figure 4.6 *Layout Using 2nd Outer Cycle*

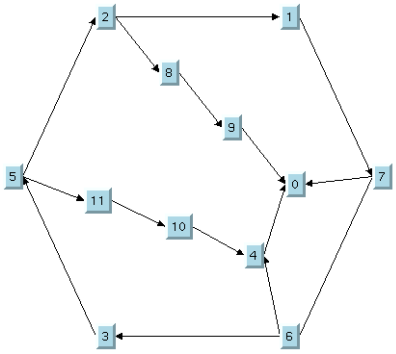


Figure 4.7 *Layout Using 3rd Outer Cycle*

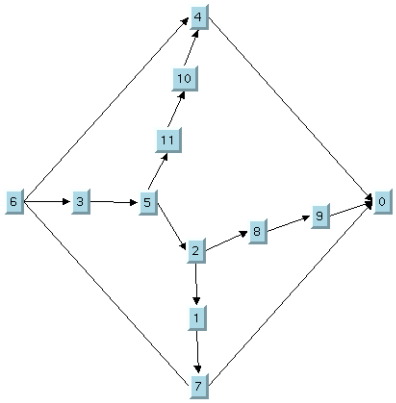


Figure 4.8 *Layout Using 4th Outer Cycle*

Refining a Graph Layout

After performing the layout on a graph, you may want to improve the quality of the layout by making some manual refinements. The following sections describe several ways to refine your layouts. When the layout is performed again after the refinements have been applied, only Step 2 of TML is redone. The results of Step 1 are reused. This is an important benefit of TML because the algorithm can recompute a layout using new parameters very quickly, without performing the time-consuming Step 1 again.

Using Fixed Nodes

One reason for applying manual refinements is to avoid overlapping nodes. To do this, you can use the fixed nodes mechanism. (See *Preserve Fixed Nodes* on page 32.)

Take a look at the original layout shown in Figure 4.9. Several overlapping nodes exist in the original layout because the nodes are concentrated in a small region and do not use the available space inside the outer cycle.

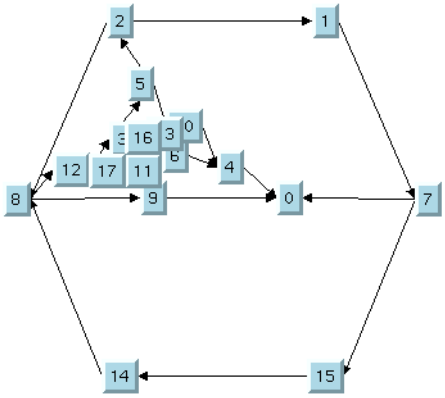


Figure 4.9 *The Original TML Layout*

To correct the problem, you can perform the following steps:

1. Move nodes 0, 9, and 10 to a place in the free space inside the outer cycle by hand as shown in Figure 4.10.

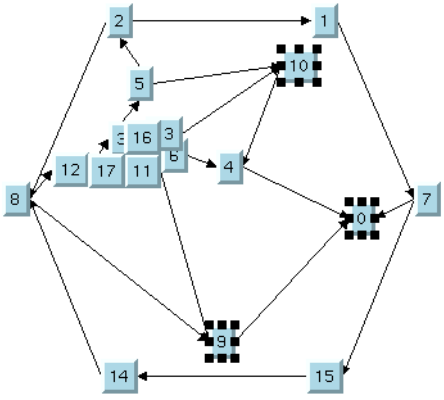


Figure 4.10 *The Layout with Nodes Moved*

2. Specify nodes 0, 9, and 10 as fixed using the `setFixed` method.
3. Use the `setPreserveFixedNodes` method to specify that the fixed nodes will not be moved when the layout is performed.
4. Perform the layout again. Only Step 2 will be performed.

   The fixed nodes “attract” the other nodes, which are distributed in the larger area inside the outer cycle as shown in Figure 4.11.

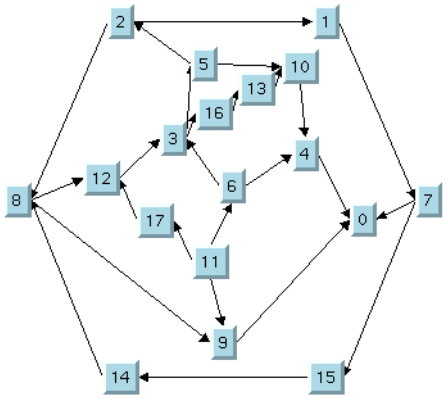


Figure 4.11 *The Final Layout*

Using the Outer Cycle Parameter

By default, Step 2 of TML produces a layout using the longest outer cycle computed in Step 1. (The length of a cycle is the number of nodes that compose the cycle.) Sometimes, a better layout can be obtained using a different choice of the outer cycle. This process of changing the outer cycle parameter and performing the layout again (see *Outer Cycle* on page 64) is a manual refinement procedure that can also be used to avoid overlapping nodes.

Note that performing the layout with a new outer cycle requires very little CPU time.

Using the Layout Region Parameter

Often, overlapping nodes can be avoided by simply increasing the size of the layout region. Figure 4.12 shows a graph drawing where several nodes overlap because the layout region is too small for the graph. Figure 4.13 shows the same graph after increasing the size of the layout region. As you can see, now there are no overlapping nodes.

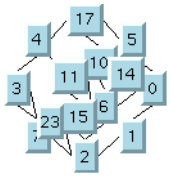


Figure 4.12 *Graph Layout: Small Layout Region and Overlapping Nodes*

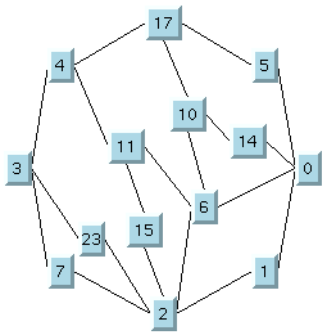


Figure 4.13 *Graph Layout: Larger Layout Region and No Overlapping Nodes*

Using a Link Clipping Interface

By default, TML does not place the connection points of links. It relies on the link connectors of the nodes to determine the connection points. If no link connectors are installed at the nodes, the default behavior is to connect to a point at the border of the bounding box of the nodes. If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape. An example is shown in Figure 4.14.

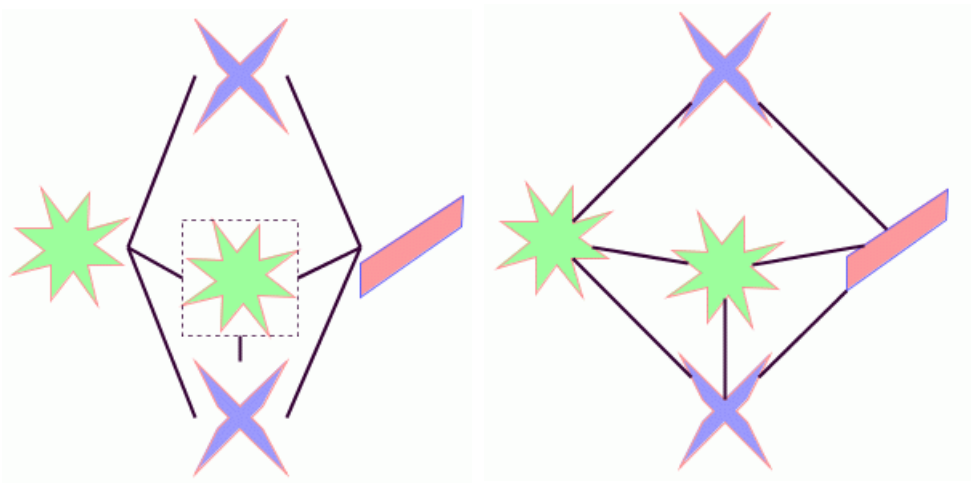


Figure 4.14 *Effect of Link Clipping Interface*

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkClipInterface`. An example for the implementation of a link clip interface is in *Link Clipping* on page 27. To set a link clip interface, call:

```
void setLinkClipInterface(IlvLinkClipInterface interface)
```

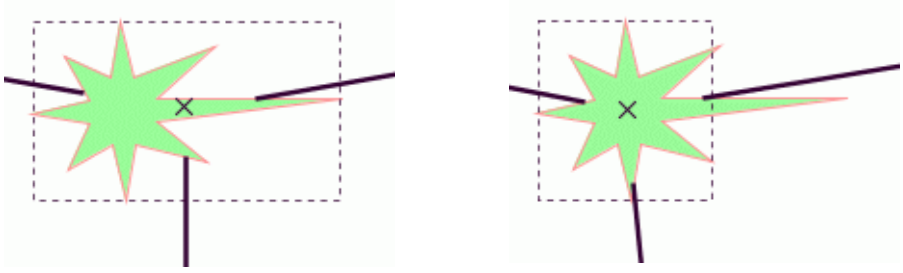
> ***Note:*** *The link clip interface requires link connectors at the nodes of an* `IlvGrapher` *that allow connector pins to be placed freely at the node border. It is recommended that you use* `IlvRelativeLinkConnector` *or* `IlvClipLinkConnector` *for link connectors to be used in combination with* `IlvGrapher` *objects. The clip link connector updates the clipped connection points automatically during interactive node movements.*

If a node has an irregular shape, the clipped links sometimes should not point towards the center of the node bounding box, but to a virtual center inside the node. You can achieve this by additionally providing a class that implements the `IlvLinkConnectionBoxInterface`. An example for the implementation of a link connection box interface is in *Link Connection Box* on page 29. To set a link connection box interface, call:

```
void setLinkConnectionBoxInterface(IlvLinkConnectionBoxInterface
interface)
```

The link connection box interface is used only when link clipping is enabled by setting a link clip interface. If no link clip interface is specified, the link connection box interface has no effect.

An example of the combined effect is shown in Figure 4.15.



Clipping at the node bounding box

Clipping at a specified connection box

Figure 4.15 *Combined Effect of Link Clipping Interface and Link Connection Box*

If the links are clipped at the green irregular star node (Figure 4.15 left), they do not point towards the center of the star, but towards the center of the bounding box of the node. This can be corrected by specifying a link connection box interface that returns a smaller node box than the bounding box (Figure 4.15 right). Alternatively, the problem could be corrected

by specifying a link connection box interface that returns the bounding box as the node box but with additional tangential offsets that shift the virtual center of the node.

Spring Embedder Layout

In this section, you will learn about the *Spring Embedder Layout* algorithm from the ILOG JViews Graph Layout module (class `IlvSpringEmbedderLayout` from the package `ilog.views.graphlayout.springembedder`).

Samples

Here are some sample drawings produced with the Spring Embedder Layout:

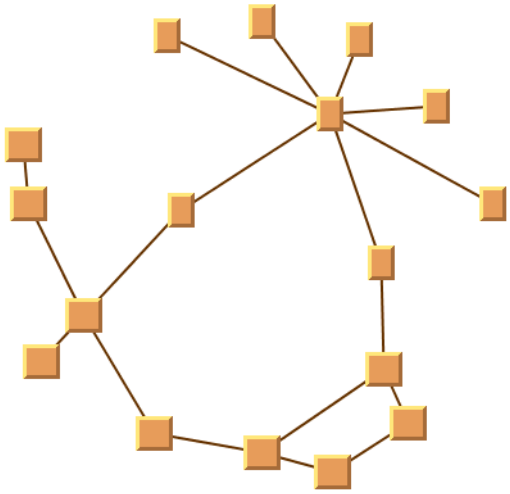


Figure 4.16 *Small Graph Drawing (Combination of Cycles and Trees) Produced with the Spring Embedder Layout*

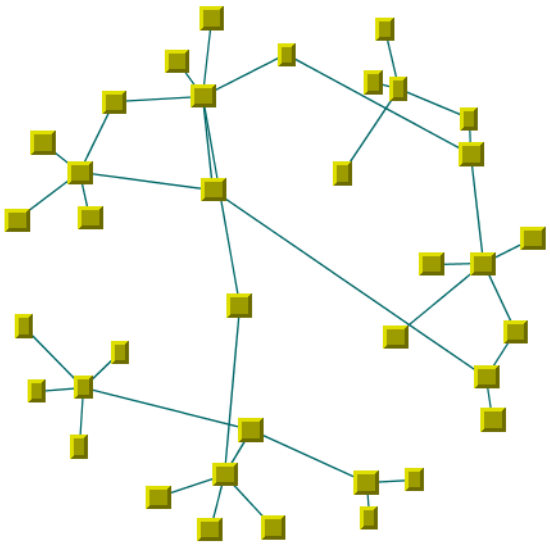


Figure 4.17 *Medium-sized Graph Drawing (Combination of Cycles and Trees) Produced with the Spring Embedder Layout*

What Types of Graphs?

Any type of graph:

- Connected and disconnected
- Planar and nonplanar graphs

Application Domains

Application domains of the Spring Embedder Layout include:

- Telecom and networking (WAN diagrams)
- Software management/software (re-)engineering (call graphs)
- CASE tools (dependency diagrams)
- Database and knowledge engineering (semantic networks, database query graphs, qualitative reasoning and other artificial intelligence diagrams, and so on)
- World Wide Web (Web hyperlink neighborhood)

Features

Often provides a drawing with no link crossings or few crossings for small- and medium-size graphs. The maximum number of nodes for which you can use this layout depends on the connectivity of the graph: hundreds for trees, dozens for cyclic graphs.

Limitations

- The computation time required to obtain an appropriate drawing grows relatively quickly with the size of the graph (that is, the number of nodes and links) and the layout process may become very time-consuming for large graphs.
- Overlapping nodes cannot always be avoided. Nevertheless, the layout algorithm often produces a drawing with no overlapping nodes.

Brief Description of the Algorithm

This algorithm implements a relatively simple version of the Spring Embedder, which is a well-known "force-directed" layout algorithm based on an analogy between graphs and mechanical systems. Nodes correspond to point masses and links correspond to springs. Attractive and repulsive forces between each pair of nodes are defined depending on the distance between the nodes and on the connectivity of the graph. The algorithm searches for a sort of minimal-energy configuration of the graph, iteratively computing the positions of the nodes.

Code Sample

Below is a code sample using the `IlvSpringEmbedderLayout` class:

```
 ...
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.springembedder.*;
 ...
IlvGrapher grapher = new IlvGrapher();
IlvManagerView view = new IlvManagerView(grapher);

 ... /*   Fill in the grapher with nodes and links here */

IlvSpringEmbedderLayout layout = new IlvSpringEmbedderLayout();
layout.attach(grapher);
try {
        IlvGraphLayoutReport layoutReport = layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
```

```
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Generic Features and Parameters

The `IlvSpringEmbedderLayout` class supports the following generic parameters defined in the `IlvGraphLayout` class:

- *Allowed Time*
- *Animation*
- *Layout of Connected Components*
- *Layout Region*
- *Link Clipping*
- *Link Connection Box*
- *Memory Savings*
- *Preserve Fixed Links*
- *Preserve Fixed Nodes*
- *Random Generator Seed Value*
- *Save Parameters to Named Properties*
- *Stop Immediately*

The following comments describe the particular way in which these parameters are used by this subclass.

Allowed Time

The layout algorithm stops if the allowed time setting has elapsed. (See *Allowed Time* on page 22.)

Animation

If animation is specified, a redraw is automatically performed after each step of the layout algorithm. (See *Animation* on page 22.)

Layout of Connected Components

The layout algorithm can utilize the generic mechanism to lay out connected components. (For more information about this mechanism, see *Layout of Connected Components* on page 25).

Layout Region

The layout algorithm uses the layout region setting (either your own or the default setting) to control the size and the position of the graph drawing. All three ways to specify the layout region are available for this subclass. (See *Layout Region* on page 25.)

Remember that if you are using the default settings, the visible area of the manager view (an instance of `IlvManagerView`) attached to the grapher is used as a layout region. If several manager views are attached, the first attached view is used. If no manager view is attached, the layout region is automatically estimated on the basis of the number and size of the nodes.

Link Clipping

The layout algorithm can use a link clip interface to clip the end points of a link. (See *Link Clipping* on page 27.)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip interface is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using a Link Clipping Interface* on page 78 for details of the link clipping mechanism.

Link Connection Box

The layout algorithm can use a link connection box interface (see *Link Connection Box* on page 29) in combination with the link clip interface. If no link clip interface is used, the link connection box interface has no effect. For details see *Using a Link Clipping Interface* on page 78.

Memory Savings

As with all classes supporting this parameter, a certain amount of memory savings can be obtained by selecting this option. (See *Memory Savings* on page 31.) Note that using this parameter will not change the resulting layout. It will just slow down the computation.

Preserve Fixed Links

The layout algorithm does not reshape the links that are specified as fixed. (See *Preserve Fixed Links* on page 32.)

Preserve Fixed Nodes

The layout algorithm does not move the nodes that are specified as fixed. (See *Preserve Fixed Nodes* on page 32.)

Random Generator Seed Value

The Spring Embedder Layout algorithm uses some randomly generated parameters. You can specify a particular value to be used as a seed value. (See *Random Generator Seed Value* on page 33.) For the default behavior, the random generator is initialized using the current system clock. Therefore, different layouts will be obtained if you perform the layout repeatedly on the same graph.

Save Parameters to Named Properties

The layout algorithm is able to save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Save Parameters to Named Properties* on page 34 and *Saving Layout Parameters and Preferred Layouts* on page 315).

Stop Immediately

The layout algorithm stops after cleanup if the method `stopImmediately` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop Immediately* on page 35.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific Parameters

The following parameters are specific to the `IlvSpringEmbedderLayout` class:

Link Style

When the layout algorithm moves the nodes, straight-line links (such as instances of `IlvLinkImage`) will automatically "follow" the new positions of their end nodes. If the grapher contains other types of links (for example, `IlvPolylineLinkImage` or `IlvSplineLinkImage`), the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can ask the layout algorithm to automatically remove all the intermediate points of the links (if any).

To do this, the following method is provided:

```
void setLinkStyle(int style)
```

The valid values for `style` are:

- `IlvSpringEmbedderLayout.NO_RESHAPE_STYLE`

  None of the links is reshaped in any manner.

- `IlvSpringEmbedderLayout.STRAIGHT_LINE_STYLE`

  All the intermediate points of the links (if any) are removed. This is the default value.

> ***Note:*** *If* `STRAIGHT_LINE_STYLE` *is selected, the layout algorithm may raise an* `IlvInappropriateLinkException` *when the layout is performed on an* `IlvGrapher`*, and links that cannot be given a straight-line shape are used. Such links are* `IlvOneLinkImage`*,* `IlvDoubleLinkImage`*,* `IlvOneSplineLinkImage`*, and* `IlvDoubleSplineLinkImage`*. You can use the method* `EnsureAppropriateLinkTypes` *or* `EnsureAppropriateLinks` *defined in* `IlvGraphLayoutUtil` *to automatically replace inappropriate links. This can be done before layout or when the* `IlvInappropriateLinkException` *is caught. For details on these methods, see the ILOG JViews Graph Layout Reference Manual. For details on the graph model, see Using the Graph Model on page 297.*

To obtain the current choice, use the following method:

```
int getLinkStyle()
```

Spring Constant

The spring constant is an important parameter because it controls the force characteristics in the spring model. Changing the value of the spring constant can cause large variations in the layout. Recommended values range from 6. to 18. A better range is 12. to 16.

To specify the spring constant, use the method:

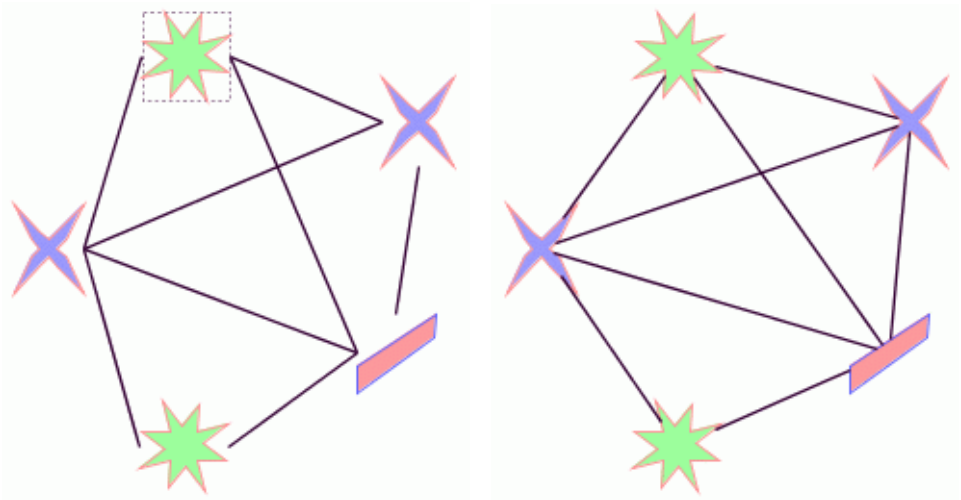
```
void setSpringConstant(float constant)
```

To obtain the current value, use the method:

```
float getSpringConstant()
```

Using a Link Clipping Interface

By default, the Spring Embedder Layout does not place the connection points of links. It relies on the link connectors of the nodes to determine the connection points. If no link connectors are installed at the nodes, the default behavior is to connect to a point at the border of the bounding box of the nodes. If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape. An example is shown in Figure 4.18.



without clipping with clipping

Figure 4.18 *Effect of Link Clipping Interface*

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkClipInterface`. An example for the implementation of a link clip interface is in *Link Clipping* on page 27. To set a link clip interface, call:

```
void setLinkClipInterface(IlvLinkClipInterface interface)
```

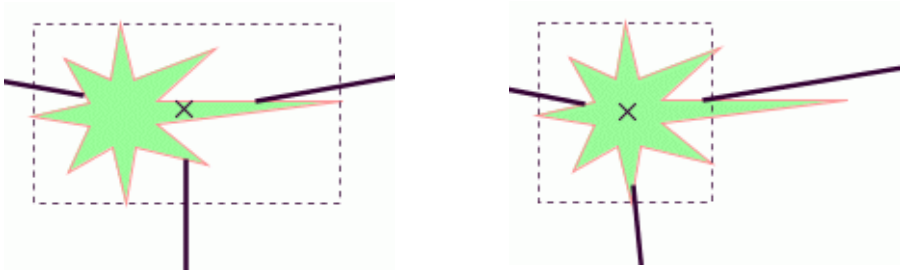
> ***Note:*** *The link clip interface requires link connectors at the nodes of an* `IlvGrapher` *that allow connector pins to be placed freely at the node border. It is recommended that you use* `IlvRelativeLinkConnector` *or* `IlvClipLinkConnector` *for link connectors to be used in combination with* `IlvGrapher` *objects. The clip link connector updates the clipped connection points automatically during interactive node movements.*

If a node has an irregular shape, the clipped links sometimes should not point towards the center of the node bounding box, but to a virtual center inside the node. You can achieve this by additionally providing a class that implements the `IlvLinkConnectionBoxInterface`. An example for the implementation of a link connection box interface is in *Link Connection Box* on page 29. To set a link connection box interface, call:

```
void setLinkConnectionBoxInterface(IlvLinkConnectionBoxInterface
interface)
```

The link connection box interface is used only when link clipping is enabled by setting a link clip interface. If no link clip interface is specified, the link connection box interface has no effect.

An example of the combined effect is shown in Figure 4.19.



Clipping at the node bounding box

Clipping at a specified connection box

Figure 4.19 *Combined Effect of Link Clipping Interface and Link Connection Box*

If the links are clipped at the green irregular star node (Figure 4.19 left), they do not point towards the center of the star, but towards the center of the bounding box of the node. This can be corrected by specifying a link connection box interface that returns a smaller node box than the bounding box (Figure 4.19 right). Alternatively, the problem could be corrected by specifying a link connection box interface that returns the bounding box as the node box but with additional tangential offsets that shift the virtual center of the node.

Uniform Length Edges Layout

In this section, you will learn about the *Uniform Length Edges Layout* algorithm from the ILOG JViews Graph Layout module (class `IlvUniformLengthEdgesLayout` from the package `ilog.views.graphlayout.uniformlengthedges`).

Samples

Here are sample drawings produced with the Uniform Length Edges Layout:

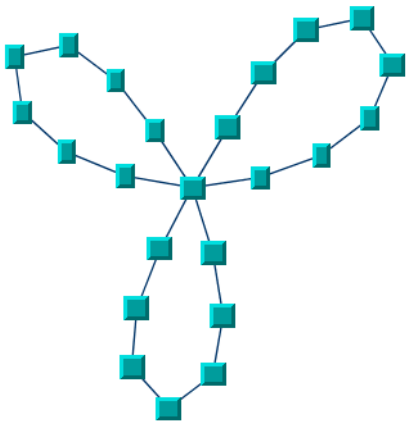


Figure 4.20 *Small Cyclic Graph Drawing Produced with the Uniform Length Edges Layout*

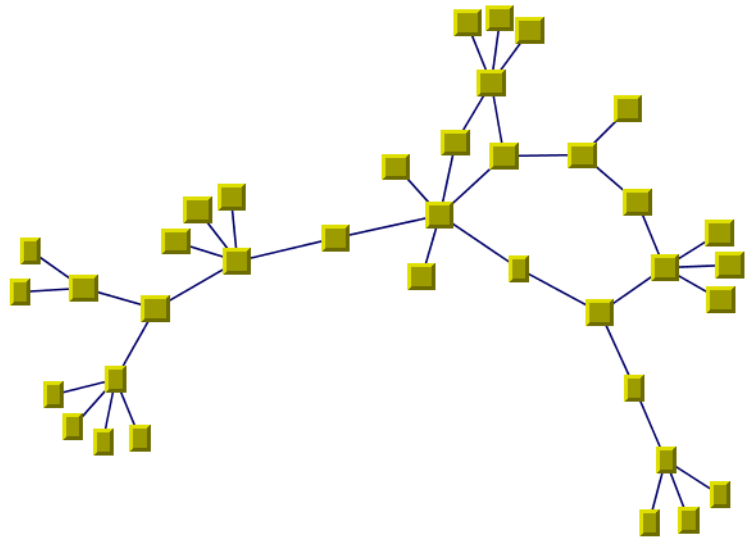


Figure 4.21 *Medium Graph Drawing (Combination of Cycles and Trees) Produced with the Uniform Length Edges Layout*

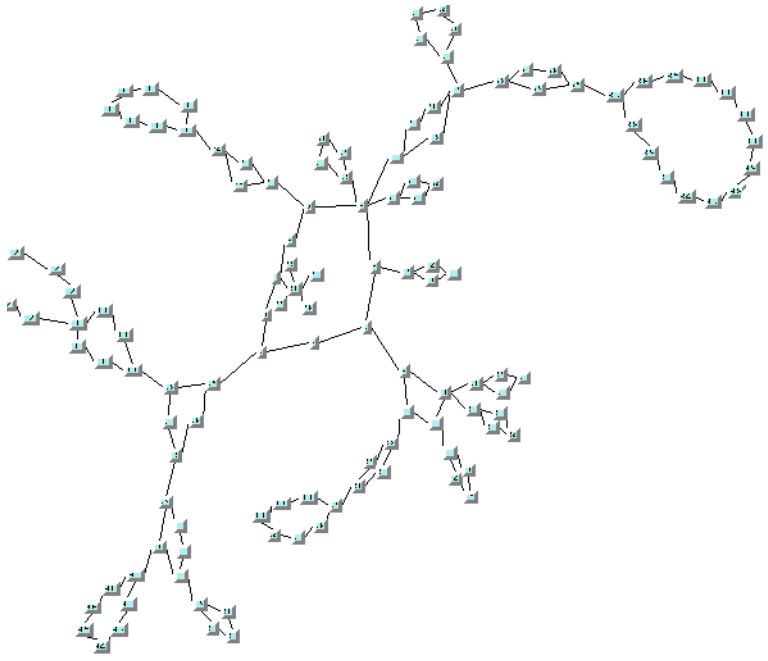


Figure 4.22 *Large Graph Drawing (Combination of Cycles and Trees) Produced with the Uniform Length Edges Layout*

What Types of Graphs?

Any type of graph:

- Connected and disconnected graphs
- Planar and nonplanar graphs

Application Domains

Application domains of the Uniform Length Edges Layout include:

- Telecom and networking (WAN diagrams)
- Software management/software (re-)engineering (call graphs)
- CASE tools (dependency diagrams)
- Database and knowledge engineering (semantic networks, database query graphs, qualitative reasoning and other artificial intelligence diagrams, and so on)
- World Wide Web (Web hyperlink neighborhood)

Features

Often provides a drawing without any or with only a few link crossings and with approximately equal length links for small- and medium-size graphs having a small number of cycles. The maximum number of nodes for which you can use the algorithm depends on the connectivity of the graph and is difficult to predict.

On demand, the algorithm can take into account the size (width and height) of the nodes. Otherwise, they are more efficiently considered as points.

It is possible to specify specific desired lengths for each of the links.

Because it takes into account the initial configuration (coordinates) of the nodes, the algorithm allows an incremental layout methodology. Sometimes the layout algorithm is not able to find the optimal layout because it has been trapped by a local minimum of the cost function. In this case, the user can perform the layout once, then move by hand one or more nodes to help the layout algorithm skip from the local minimum, and then perform the layout again. However, the success of this procedure depends on the choice of the nodes to be moved. In many cases, the choice is quite intuitive.

Limitations

- The algorithm is not appropriate for all graphs. In particular, it will produce bad results on some highly connected cyclic graphs for which a planar drawing with equal-length links may simply not exist.

- The computation time required to obtain an appropriate drawing grows relatively quickly with the size of the graph (that is, the number of nodes and links) and the layout process may become time-consuming for large graphs.
- Overlapping nodes cannot always be avoided. Nevertheless, the layout algorithm often produces a drawing with no overlapping nodes.

Brief Description of the Algorithm

This layout algorithm iteratively searches for a configuration of the graph where the length of the links is close to a user-defined or a default value.

Code Sample

Below is a code sample using the `IlvUniformLengthEdgesLayout` class:

```
 ...
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.uniformlengthedges.*;
 ...
IlvGrapher grapher = new IlvGrapher();
IlvManagerView view = new IlvManagerView(grapher);

 ... /*   Fill in the grapher with nodes and links here */

IlvUniformLengthEdgesLayout layout = new
                                        IlvUniformLengthEdgesLayout();
layout.attach(grapher);
try {
        IlvUniformLengthEdgesLayoutReport layoutReport =
                layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Generic Features and Parameters

The `IlvUniformLengthEdgesLayout` class supports the following generic parameters defined in the `IlvGraphLayout` class:

- *Allowed Time*
- *Animation*

- *Layout of Connected Components*
- *Layout Region*
- *Link Clipping*
- *Link Connection Box*
- *Preserve Fixed Links*
- *Preserve Fixed Nodes*
- *Save Parameters to Named Properties*
- *Stop Immediately*

The following comments describe the particular way in which these parameters are used by this subclass.

Allowed Time

The layout algorithm stops if the allowed time setting has elapsed. (See *Allowed Time* on page 22.)

Animation

If animation is specified, a redraw is automatically performed after each step of the layout algorithm. (See *Animation* on page 22.)

Layout of Connected Components

The layout algorithm can utilize the generic mechanism to lay out connected components. (For more information about this mechanism, see *Layout of Connected Components* on page 25.)

Layout Region

The layout algorithm can use the layout region setting (either your own or the default setting) to control the size and the position of the graph drawing. All three ways to specify the layout region are available for this subclass. (See *Layout Region* on page 25.)

Note that by default the Uniform Length Edges Layout algorithm does not use the layout region. (For details see also *Force Fit to Layout Region* on page 88.)

Remember that if you are using the default settings, the visible area of the manager view (an instance of `IlvManagerView`) attached to the grapher is used as a layout region. If several manager views are attached, the first attached view is used. If no manager view is attached, the layout region is automatically estimated on the basis of the number and size of the nodes.

Link Clipping

The layout algorithm can use a link clip interface to clip the end points of a link. (See *Link Clipping* on page 27.)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip interface is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using a Link Clipping Interface* on page 91 for details of the link clipping mechanism.

Link Connection Box

The layout algorithm can use a link connection box interface (see *Link Connection Box* on page 29) in combination with the link clip interface. If no link clip interface is used, the link connection box interface has no effect. For details see *Using a Link Clipping Interface* on page 91.

Preserve Fixed Links

The layout algorithm does not reshape the links that are specified as fixed. (See *Preserve Fixed Links* on page 32. See also *Link Style* on page 86.)

Preserve Fixed Nodes

The layout algorithm does not move the nodes that are specified as fixed. Moreover, the algorithm takes into account the fixed nodes when computing the position of the nonfixed nodes. (See *Preserve Fixed Nodes* on page 32.)

Save Parameters to Named Properties

The layout algorithm is able to save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Save Parameters to Named Properties* on page 34 and *Saving Layout Parameters and Preferred Layouts* on page 315).

Stop Immediately

The layout algorithm stops after cleanup if the method `stopImmediately` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop Immediately* on page 35.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific Parameters

The following parameters are specific to the `IlvUniformLengthEdgesLayout` class.

Link Style

When the layout algorithm moves the nodes, straight-line links (such as instances of `IlvLinkImage`) will automatically “follow” the new positions of their end nodes. If the grapher contains other types of links (for example, `IlvPolylineLinkImage` or `IlvSplineLinkImage`), the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can ask the layout algorithm to automatically remove all the intermediate points of the links (if any).

To do this, the following method is provided:

```
void setLinkStyle(int style)
```

The valid values for `style` are:

- `IlvUniformLengthEdgesLayout.NO_RESHAPE_STYLE`

  None of the links is reshaped in any manner.

- `IlvUniformLengthEdgesLayout.STRAIGHT_LINE_STYLE`

  All the intermediate points of the links (if any) are removed. This is the default value.

> ***Note:*** *If* `STRAIGHT_LINE_STYLE` *is selected, the layout algorithm may raise an* `IlvInappropriateLinkException` *when the layout is performed on an* `IlvGrapher` *and links that cannot be given a straight-line shape are used. Such links are* `IlvOneLinkImage`, `IlvDoubleLinkImage`, `IlvOneSplineLinkImage`, *and* `IlvDoubleSplineLinkImage`. *You can use the method* `EnsureAppropriateLinkTypes` *or* `EnsureAppropriateLinks` *defined in* `IlvGraphLayoutUtil` *to automatically replace inappropriate links. This can be done before layout or when the* `IlvInappropriateLinkException` *is caught. For details on these methods, see the ILOG JViews Graph Layout Reference Manual. For details on the graph model, see Using the Graph Model on page 297.*

To obtain the current choice, use the following method:

```
int getLinkStyle()
```

Number of Iterations

The iterative computation of the layout algorithm is stopped if the time exceeds the allowed time (see *Allowed Time* on page 22) or if the number of iterations exceeds the allowed number of iterations. To specify this number, use the method:

```
void setAllowedNumberOfIterations(int iterations)
```

To obtain the current value, use the method:

```
int getAllowedNumberOfIterations()
```

Preferred Length

The main objective of this layout algorithm is to obtain a layout where all the links have a given length. This is called the “preferred length.” To specify the preferred length, use the method:

```
void setPreferredLinksLength(float length)
```

To obtain the current value, use the method:

```
float getPreferredLinksLength()
```

It is also possible to specify a desired length for individual links, using the method:

```
void setPreferredLength(Object link, float length)
```

To obtain the current value, use the method:

```
float getPreferredLength(Object link)
```

If a specific length is not specified for a link, the global settings are used.

Respect Node Size

By default, the layout algorithm ignores the size (width and height) of the nodes. For efficiency reasons, the nodes are approximated with points place in the center of the bounding box of the nodes. When dealing with large nodes, the preferred length parameter can be increased in such a way that the nodes do not overlap.

However, to improve the support for graphs with heterogeneous node sizes, the algorithm provides a special mode in which the particular size of each node is taken into consideration.

To set and retrieve this mode, use the following methods:

```
void setRespectNodeSizes(boolean respect)
```

```
boolean isRespectNodeSizes()
```

Force Fit to Layout Region

For this layout algorithm, it is more difficult than for others to choose an appropriate size for the layout region. If the specified layout region is too small for a given graph, the resulting layout will not be the best. For this reason, by default, the Uniform Length Edges Layout algorithm does not use the layout region parameter. It can use as much space as it needs to lay out the grapher.

To specify whether the layout algorithm must use the layout region, use the following method:

```
void setForceFitToLayoutRegion(boolean option)
```

To obtain the current value, use the following method:

```
boolean isForceFitToLayoutRegion()
```

The default value of the parameter is `false`.

For Experts: Additional Features

The following expert features are available.

Maximum Allowed Move Per Iteration

At each iteration, the layout algorithm moves the nodes a relatively small amount. This amount should not be too large; otherwise the algorithm may not converge. But it should not

be too small either, otherwise the number of necessary iterations increases and the running time does also.

The maximum amount of movement at each iteration is controlled by a parameter that can be set and retrieved with the following methods:

```
void setMaxAllowedMovePerIteration(float maxMove)
float getMaxAllowedMovePerIteration()
```

Link Length Weight

The layout algorithm is based on the computation of attraction and repulsion forces for each of the nodes and the iterative search of an equilibrum configuration. One of these forces is related to the objective of obtaining a link length close to the specified preferred length. The weight of this force, representing the total amount of forces, is controlled by a parameter that can be set and retrieved with the following methods:

```
void setLinkLengthWeight(float weight)
float getLinkLengthWeight()
```

The default value is `1`. Increasing this parameter can help obtain link lengths closer to the specified length, but increasing too much can increase the number of link crossings.

Additional Node Repulsion Weight

An additional repulsion force can be computed between nodes that are not connected by a link. The weight of this force, representing the total amount of forces, is controlled by a parameter that can be set and retrieved using the following methods:

```
void setAdditionalNodeRepulsionWeight(float weight)
float getAdditionalNodeRepulsionWeight()
```

The default value of this parameter is `0`, that is, by default this force is not computed.

Setting this weight to some small values, for instance `0.2` (that is one-fifth of the default link length weight), can help avoid overlaps between nodes that are not connected by links.

The following pictures allow you to compare the same graph laid out with the additional repulsion disabled (Figure 4.23) and then enabled (Figure 4.24). You can see that the “star” configuration, where many nodes are connected to the same central node, is better displayed when the additional repulsion is enabled.

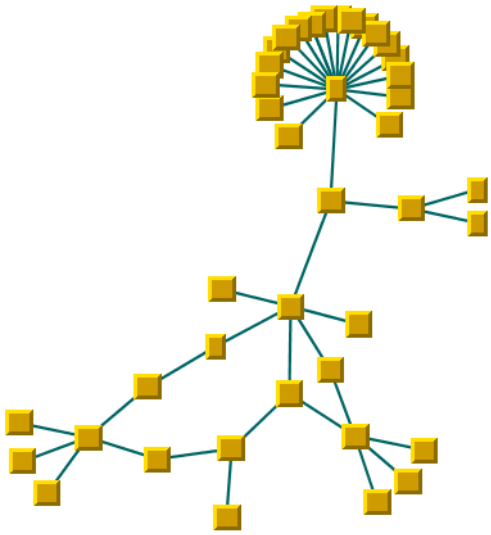


Figure 4.23 *Additional Repulsion Disabled, Produced with the Uniform Length Edges Layout*

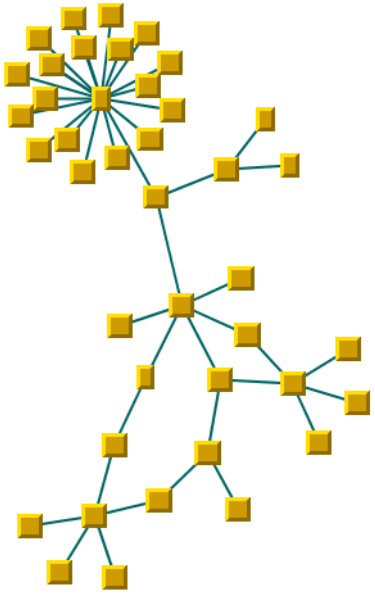


Figure 4.24 *Additional Repulsion Enabled, Produced with the Uniform Length Edges Layout*

Sometimes, for large graphs, better results are obtained if the layout is performed once without the additional force (this may help for the convergence of the algorithm), then a second time with the additional force (as a refinement step).

Node Distance Threshold

The additional repulsion force between two nodes not connected by a link is computed only when their distance is smaller than a predefined distance. This distance can be set and retrieved using the following methods:

```
void setNodeDistanceThreshold(float threshold)
float getNodeDistanceThreshold()
```

Note that this additional force is computed only if the "additional node distance weight" is set to a value larger than `0` (which is the default).

It is recommended that this threshold be set to a value smaller than the preferred length of the links.

Using a Link Clipping Interface

By default, the Uniform Length Edges Layout does not place the connection points of links. It relies on the link connectors of the nodes to determine the connection points. If no link connectors are installed at the nodes, the default behavior is to connect to a point at the border of the bounding box of the nodes. If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape. An example is shown in Figure 4.25.

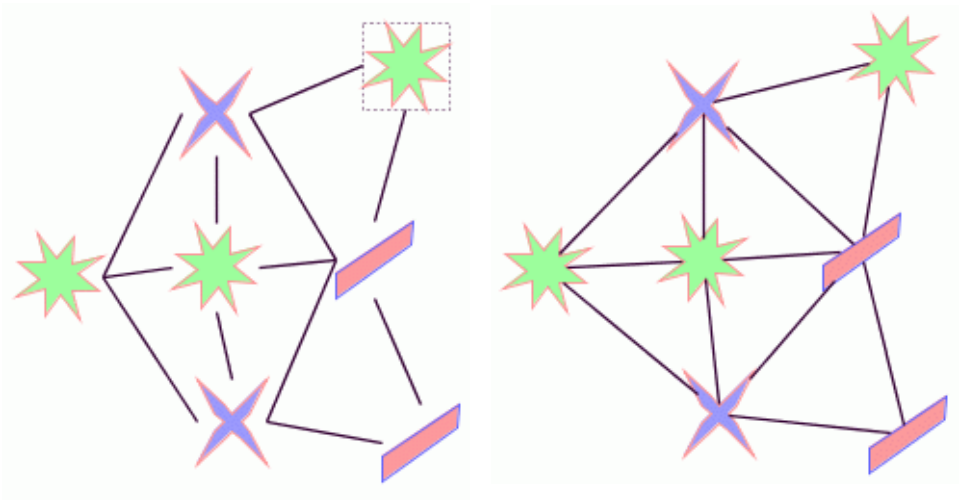


Figure 4.25 *Effect of Link Clipping Interface*

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkClipInterface`. An example for the implementation of a link clip interface is in *Link Clipping* on page 27. To set a link clip interface, call:

```
void setLinkClipInterface(IlvLinkClipInterface interface)
```

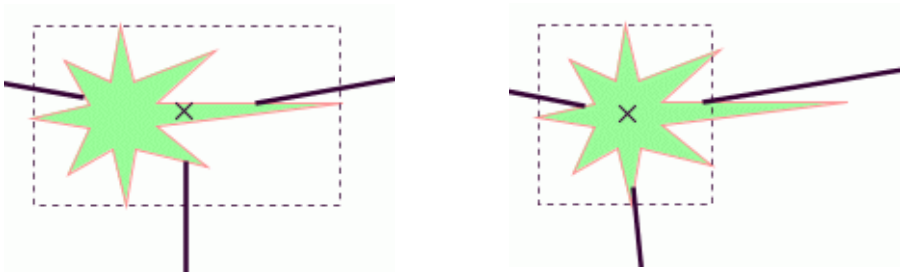
> ***Note:*** *The link clip interface requires link connectors at the nodes of an* `IlvGrapher` *that allow connector pins to be placed freely at the node border. It is recommended that you use* `IlvRelativeLinkConnector` *or* `IlvClipLinkConnector` *for link connectors to be used in combination with* `IlvGrapher` *objects. The clip link connector updates the clipped connection points automatically during interactive node movements.*

If a node has an irregular shape, the clipped links sometimes should not point towards the center of the node bounding box, but to a virtual center inside the node. You can achieve this by additionally providing a class that implements the `IlvLinkConnectionBoxInterface`. An example for the implementation of a link connection box interface is in *Link Connection Box* on page 29. To set a link connection box interface, call:

```
void setLinkConnectionBoxInterface(IlvLinkConnectionBoxInterface
interface)
```

The link connection box interface is used only when link clipping is enabled by setting a link clip interface. If no link clip interface is specified, the link connection box interface has no effect.

An example of the combined effect is shown in Figure 4.26.



Clipping at the node bounding box

Clipping at a specified connection box

Figure 4.26 *Combined Effect of Link Clipping Interface and Link Connection Box*

If the links are clipped at the green irregular star node (Figure 4.26 left), they do not point towards the center of the star, but towards the center of the bounding box of the node. This can be corrected by specifying a link connection box interface that returns a smaller node box than the bounding box (Figure 4.26 right). Alternatively, the problem could be corrected by specifying a link connection box interface that returns the bounding box as the node box but with additional tangential offsets that shift the virtual center of the node.

Tree Layout

In this section, you will learn about the *Tree Layout* algorithm from the ILOG JViews Graph Layout module (class `IlvTreeLayout` from the package `ilog.views.graphlayout.tree`).

Samples

Here are some sample drawings produced with the Tree Layout:

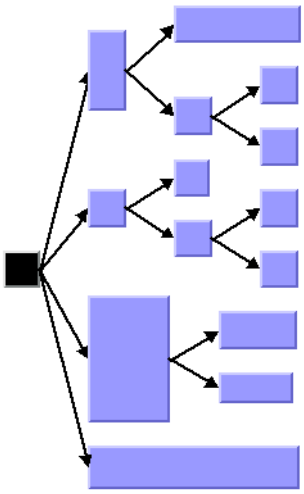


Figure 4.27 *Tree Layout in Free Layout Mode with Center Alignment and Flow Direction to the Right*

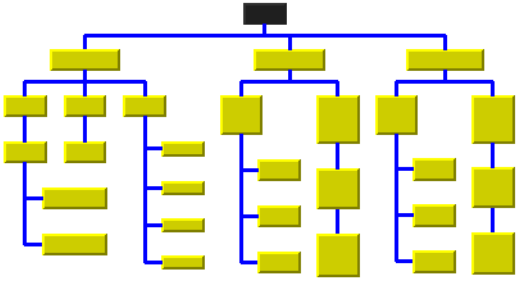


Figure 4.28 *Tree Layout with Flow Direction to the Bottom, Orthogonal Link Style, and Tip-Over Alignment at Some Leaf Nodes*

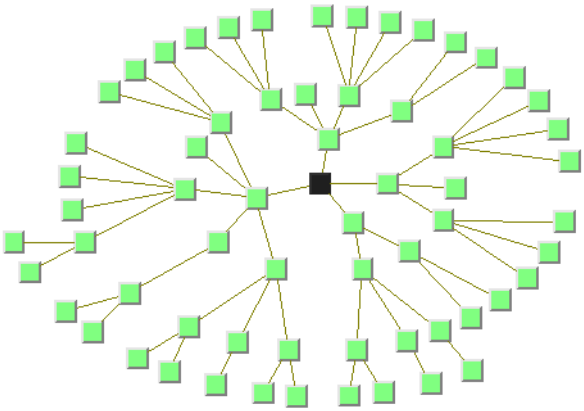


Figure 4.29 *Tree Layout in Radial Layout Mode with Aspect Ratio 1.5*

What Types of Graphs?

- Primarily designed for pure trees. It can also be used for non-trees, that is, for cyclic graphs. In this case, the algorithm computes and uses a spanning tree of the graph, ignoring all links that do not belong to the spanning tree.
- Directed and undirected trees. If the links are directed, the algorithm automatically chooses the canonical root node. If the links are undirected, you can choose a root node.
- Connected and disconnected graphs. If the graph is not connected, the layout algorithm treats each connected component separately. Each component has exactly one root node. In this case, a forest of trees is laid out.

Application Domains

Application domains of the Tree Layout include:

- Business processing (organizational charts)
- Software management/software (re-)engineering (UML diagrams, call graphs)
- Database and knowledge engineering (decision trees)
- The World Wide Web (Web site maps)

Features

- Takes into account the size of the nodes so that no overlapping occurs.

- Optionally, reshapes the links to give them an orthogonal form (alternating horizontal and vertical line segments).
- Various layout modes: *free*, *levels*, *radial*, or *automatic tip-over.*
  - In the free layout mode, arranges the children of each node, starting recursively from the root, so that the links flow uniformly in the same direction.
  - In the level layout mode, partitions the nodes into levels, and arranges the levels horizontally or vertically.
  - In radial layout mode, partitions the nodes into levels, and arranges the levels in circles or ellipses around the root.
  - In the tip-over modes, arranges the nodes in a similar way to the free layout mode, but tries to tip over children automatically to better fit the layout to the given aspect ratio.
- Provides several alignment and offset options.
- Allows you to specify nodes that must be directly neighbored.
- Provides incremental and nonincremental modes. Incremental mode takes the previous position of nodes into account and positions the nodes without changing the relative order of the nodes in the tree so that the layout is stable on incremental changes of the graph.
- Very efficient, scalable algorithm. Produces a nice layout quickly even if the number of nodes is huge.

Limitations

- If the orthogonal setting is not specified as the link style (see *Link Style* on page 109), some links may in rare cases overlap nodes depending on the size of the nodes, the alignment parameters, and the offset parameters.
- The layout algorithm first determines a spanning tree of the graph. If the graph is not a pure tree, some links will not be included as part of the spanning tree. These links are ignored. For this reason, they may cross other links or overlap nodes in the final layout.
- For stability in incremental mode, the algorithm tries to preserve the relative order of the children of each node. It uses a heuristic to calculate the relative order from the previous positions of the nodes. The heuristic may fail if children overlap their old positions or are not aligned horizontally or vertically.
- Despite preserving the relative order of the children, in rare cases the layout is not perfectly stable in incremental radial layouts. Subsequent layouts may rotate the nodes around the root, although the relative circular order of the nodes within their circular levels is still preserved.

- The tip-over layout modes will perform several trial layouts with different tip-over alignment options according to various heuristics. From these trial layouts, the algorithm picks the layout that best fits the given aspect ratio. This may not be the optimal layout for the aspect ratio, but it is the best layout among the trials. Calculating the absolute best-fitting layout is not feasible computationally (it is generally an NP-complete problem).

Brief Description of the Algorithm

The core algorithm for the free, level, and radial layout modes works in just two steps and is very fast.

The variations of the tip-over layout mode perform the second step several times and pick the layout result that best fits the given aspect ratio (the ratio between width and height of the drawing area). For this reason, the tip-over layout modes are slower.

Step 1: Calculating the Spanning Tree

If the graph is disconnected, the layout algorithm chooses a *root node* for each connected component. Starting from the root node, it traverses the graph to choose the links of the *spanning tree*. If the graph is a pure tree, all links are chosen. If the graph has cycles, some links will not be included as part of the spanning tree. These links are called *non-tree links*, while the links of the spanning tree are called *tree links*. The non-tree links are ignored in step 2 of the algorithm.

In Figure 4.27, Figure 4.28, and Figure 4.29,the *root* is the black node. In the spanning tree, each node except the root has a parent node. All nodes that have the same parent are called children with respect to the parent and siblings with respect to themselves. Nodes without children are called leaves. Each child at a node starts a subtree (also called a branch of the tree). Figure 4.30 show an example of a spanning tree.

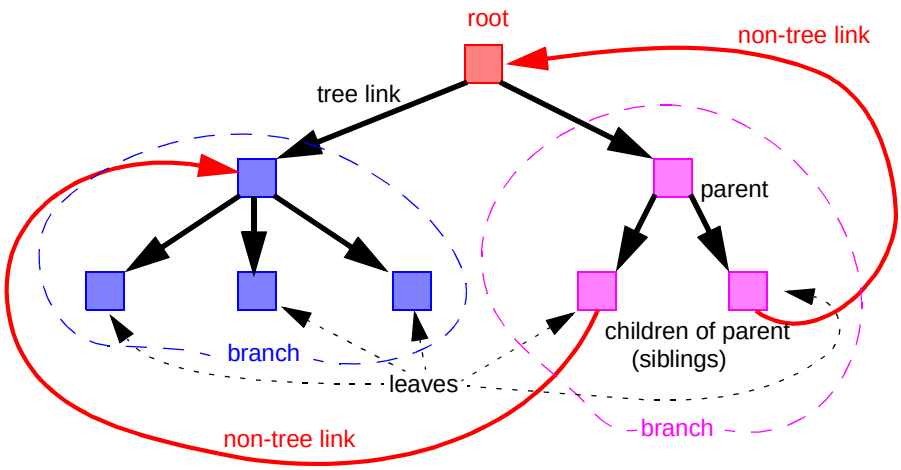


Figure 4.30 *A Spanning Tree*

Step 2: Calculating Node Positions and Link Shapes

The layout algorithm arranges the nodes according to the layout mode and the offset and alignment options. In the free mode and level mode, the nodes are arranged horizontally or vertically so that all tree links flow roughly in the same direction. In the radial layout modes, the nodes are arranged in circles or ellipses around the root so that all tree links flow radially away from the root. Finally, the link shapes are calculated according to the link style and alignment options.

Code Sample

Below is a code sample using the `IlvTreeLayout` class:

```
...
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.tree.*;
 ...
IlvGrapher grapher = new IlvGrapher();
IlvManagerView view = new IlvManagerView(grapher);

 ... /* Fill in the grapher with nodes and links here */
 ... /* Suppose we have added rootNode as a node in the grapher */

IlvTreeLayout layout = new IlvTreeLayout();
layout.attach(grapher);

/* Specify the root node, orientation and alignment */
```

```
layout.setRoot(rootNode);
layout.setFlowDirection(IlvDirection.Right);
layout.setGlobalAlignment(IlvTreeLayout.CENTER);

try {
        IlvGraphLayoutReport layoutReport = layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Generic Features and Parameters

The `IlvTreeLayout` class supports the following generic features defined in the `IlvGraphLayout` class:

- *Allowed Time*
- *Layout of Connected Components*
- *Link Clipping*
- *Link Connection Box*
- *Percentage Completion Calculation*
- *Preserve Fixed Links*
- *Preserve Fixed Nodes*
- *Save Parameters to Named Properties*
- *Stop Immediately*

The following paragraphs describe the particular way in which these features are used by this subclass.

Allowed Time

The layout algorithm stops if the allowed time setting has elapsed. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Allowed Time* on page 22.) If the layout stops early because the allowed time has elapsed, the nodes and links are not moved from their positions before the layout call and the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Layout of Connected Components

The layout algorithm can utilize the generic mechanism to layout connected components. (For more information about this mechanism, see *Layout of Connected Components* on page 25). It has, however, a specialized internal mechanism to layout connected components and, therefore, the generic mechanism is switched off by default.

The generic connected component layout mechanism has the disadvantage that it moves connected components completely. Fixed nodes within a component do not preserve their old position, and the resulting layout may be unstable on incremental changes, depending on which layout instance is used for the component layout.

If the generic connected component layout mechanism is disabled, the algorithm uses its own specialized internal mechanism instead of the generic mechanism to lay out each component as a separate tree. This is usually faster and more stable on incremental changes than the generic mechanism. Furthermore, it enables the user to set the position of the layout.

Link Clipping

The layout algorithm can use a link clip interface to clip the end points of a link. (See *Link Clipping* on page 27.)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip interface is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using a Link Clipping Interface* on page 114 for details of the link clipping mechanism.

Link Connection Box

The layout algorithm can use a link connection box interface (see *Link Connection Box* on page 29) in combination with the link clip interface. If no link clip interface is used, the link connection box interface has no effect. For details see *Using a Link Connection Box Interface* on page 113.

Percentage Completion Calculation

The layout algorithm calculates the estimated percentage of completion. This value can be obtained from the layout report during the run of layout. (For a detailed description of this feature, see *Percentage Completion Calculation* on page 31 and *Layout Event Listeners* on page 294.)

Preserve Fixed Links

The layout algorithm does not reshape the links that are specified as fixed. (For more information on link parameters in the `IlvGraphLayout` class, see *Preserve Fixed Links* on page 32 and *Link Style* on page 109.)

Preserve Fixed Nodes

The layout algorithm does not move the nodes that are specified as fixed. (For more information on node parameters in the `IlvGraphLayout` class, see *Preserve Fixed Nodes* on page 32.) Moreover, the layout algorithm ignores fixed nodes completely and also does not route the links that are incident to the fixed nodes. This can result in undesired overlapping nodes and link crossings. However, this feature is useful for individual, disconnected components that can be laid out independently.

Save Parameters to Named Properties

The layout algorithm can save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Save Parameters to Named Properties* on page 34 and *Saving Layout Parameters and Preferred Layouts* on page 315.)

Stop Immediately

The layout algorithm stops after cleanup if the method `stopImmediately` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop Immediately* on page 35.) If the layout stops early because the allowed time has elapsed, the nodes and links are not moved from their positions before the layout call, and the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific Parameters (All Tree Layout Modes)

The following parameters are specific to the `IlvTreeLayout` class. They apply to all layout modes.

Root Node

The final layout is influenced mainly by the choice of the root node.

The root node is placed in a prominent position. For instance, in a top-down drawing with free layout mode, it is placed at the top of the tree. With the radial layout mode, it is placed at the center of the tree.

The spanning tree is calculated starting from the root node. If the graph is disconnected, the layout algorithm needs one root node for each connected component.

The layout algorithm automatically selects a root node when needed. It uses a heuristic that calculates preferences for all nodes to become a root. It chooses the node with the highest preference. The heuristic gives nodes without incoming links the highest preference and leaf nodes without outgoing links the lowest preference. Hence, in a directed tree, the canonical root is always chosen automatically.

It is possible to influence the choice of the root node. To select a node explicitly as the root, use the method:

```
void setRoot(Object node);
```

This gives the node the maximal preference to become the root during layout. If only one node is specified this way, the algorithm selects this node. If several nodes of the same connected component are specified this way, the layout algorithm chooses one of them as the root.

For Experts: Additional Options for Root Nodes

The layout algorithm manages a list of the root nodes that have been specified by the `setRoot` method. To obtain the nodes in this list, use the following method:

```
Enumeration getSpecRoots();
```

After layout, you can also retrieve the list of root nodes that were actually used by the algorithm. This list is not necessarily the same as the list of specified roots. For instance, it contains the chosen root nodes if none were specified or if too many were specified. To obtain the root nodes that were used by the algorithm, use the following method:

```
Enumeration getCalcRoots();
```

The following example shows how to iterate over the calculated root nodes and print the root node preferences:

```
Enumeration e = layout.getCalcRoots();
while (e.hasMoreElements()) {
  node = e.nextElement();
  System.out.println("Preference:" + layout.getRootPreference(node));
}
```

To directly manipulate the root node preference value of an individual node, you can use the following method:

```
setRootPreference(Object node, int preference);
```

In this case, the layout uses the specified value instead of the heuristically calculated preference for the node. The normal preference value should be between `0` and `10000`. Specifying a root node explicitly corresponds to setting the preference value to `10000`. If you want to prohibit a node from becoming the root, specify a preference value of zero (`0`).

A negative preference value indicates that the layout algorithm should recalculate the root node preference using the heuristic. If a root was specified by the `setRoot` method but this node should no longer be the root in subsequent layouts, use the following call to clear the root node setting:

```
layout.setRootPreference(node, -1);
```

This call also removes the node from the list of specified roots.

Position Parameters

To set the position of the *top left corner* of the layout to (10, 10) use the following call:

```
layout.setPosition(new IlvPoint(10, 10), true);
```

If the graph consists of only a single tree, it is often more convenient to set the position of the root node instead. This can be done by the same method, passing `false` instead of `true`:

```
layout.setPosition(point, false);
```

If no position is specified, the layout keeps the root node at its previous position.

To obtain the current position, use the following method:

```
IlvPoint getPosition();
```

This method returns `null` if no position was specified.

If the method returns a point, use the following method to query whether the specified position is the top left corner or the position of the first root node:

```
boolean isRootPosition();
```

Using Compass Directions for Situating Layout Parameters

To simplify the explanations of the layout parameters, we use the compass directions *north, south, east* and *west*. The center of the root node of a tree is considered the north pole.

In the nonradial layout modes, the link flow direction always corresponds to south. If the root node is placed at the top of the drawing, north is at the top, south at the bottom, east to the right, and west to the left. If the root node is placed at the left border of the drawing, north is to the left, south to the right, east at the top, and west at the bottom.

In the radial layout modes, the root node is placed in the center of the drawing. The meaning of north and south depends on the position relative to the root: the north side of the node is the side closer to the root and the south side is the side further away from the root. The east direction is counterclockwise around the root and the west direction is clockwise around the root. This is similar to a cartographic map of a real globe that shows the area of the north pole as if you were looking down at the top of the globe.

Compass directions are used to provide uniform naming conventions for certain layout options. They occur in the alignment options, the level alignment option, and the east-west neighboring feature, which are explained later. In Figure 4.31 and Figure 4.45, the compass icons show the compass directions in these drawings.

Layout Modes

The tree layout algorithm has several layout modes. To select a layout mode, use the method:

```
void setLayoutMode(int mode);
```

To obtain the current layout mode, call:

```
int getLayoutMode();
```

The available layout modes are the following:

- `IlvTreeLayout.FREE` (the default)

- `IlvTreeLayout.LEVEL`
- `IlvTreeLayout.RADIAL`
- `IlvTreeLayout.ALTERNATING_RADIAL`
- `IlvTreeLayout.TIP_OVER`
- `IlvTreeLayout.TIP_ROOTS_OVER`
- `IlvTreeLayout.TIP_LEAVES_OVER`
- `IlvTreeLayout.TIP_ROOTS_AND_LEAVES_OVER`

The following sections describe the characteristics and the layout parameters of each layout mode.

Free Layout Mode

The free layout mode arranges the children of each node starting recursively from the root so that the links flow roughly in the same direction. For instance, if the link flow direction is top-down, the root node is placed at the top of the drawing. Siblings (nodes that have the same parent) are justified at their top borders, but nodes of different tree branches (nodes with different parents) are not justified.

The following statement sets the free layout mode:

```
layout.setLayoutMode(IlvTreeLayout.FREE);
```

Flow Direction

The flow direction parameter specifies the direction of the tree links. The compass icons show the compass directions in these layouts.

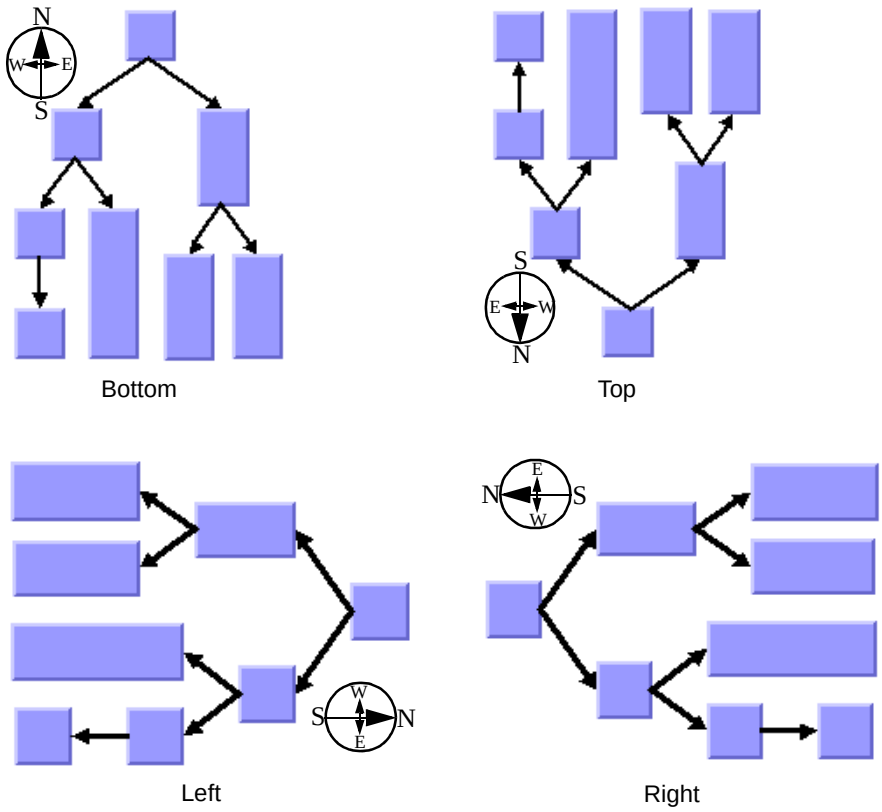


Figure 4.31 *Flow Directions*

If the flow direction is to the bottom, the root node is placed topmost. Each parent node is placed above its children, which are normally arranged horizontally. (This tip-over alignment is an exception.)

If the flow direction is to the right, the root node is placed leftmost. Each parent node is placed to the left of its children, which are normally arranged vertically.

To specify the flow direction, use the following method:

```
void setFlowDirection(int direction);
```

The valid values for `direction` are:

- `IlvDirection.Right` (the default)
- `IlvDirection.Left`
- `IlvDirection.Bottom`

- `IlvDirection.Top`

To obtain the current setting, use the following method:

```
int getFlowDirection();
```

Alignment Parameter

The alignment option controls how a parent is placed relative to its children. The alignment can be set globally, in which case all nodes are aligned in the same way, or locally on each node, with the result that different alignments occur in the same drawing.

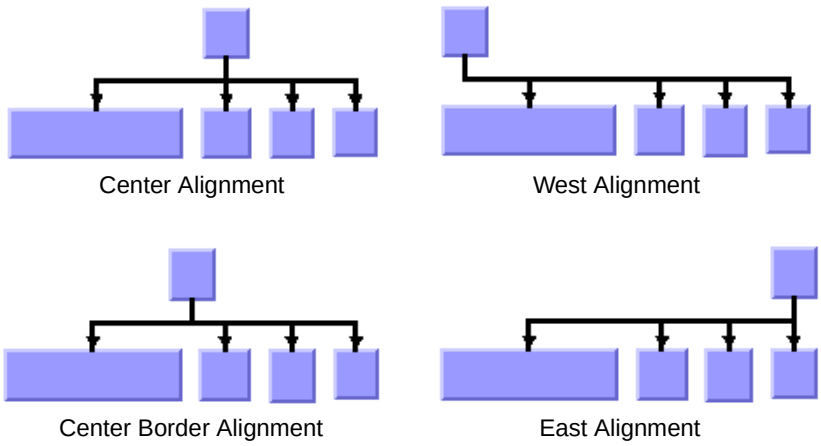


Figure 4.32 *Alignment Options*

Global Alignment Parameters

To set the global alignment, use the following method:

```
void setGlobalAlignment(int alignment);
```

The valid values for `alignment` are:

- `IlvTreeLayout.CENTER` (the default)

  The parent is centered over its children, taking the center of the children into account.

- `IlvTreeLayout.BORDER_CENTER`

  The parent is centered over its children, taking the border of the children into account. If the size of the first and the last child varies, the border center alignment places the parent closer to the larger child than to the default center alignment.

- `IlvTreeLayout.EAST`

  The parent is aligned with the border of its easternmost child. For instance, if the flow direction is to the bottom, east is the direction to the right. If the flow direction is to the

top, east is the direction to the left. See *Using Compass Directions for Situating Layout Parameters* on page 103 for details.

- `IlvTreeLayout.WEST`

  The parent is aligned with the border of its westernmost child. For instance, if the flow direction is to the bottom, west is the direction to the left. If the flow direction is to the right, west is the direction to the bottom. See *Using Compass Directions for Situating Layout Parameters* on page 103 for details.

- `IlvTreeLayout.TIP_OVER`

  The children are arranged sequentially instead of in parallel, and the parent node is placed with an offset to the children. For details see *Tip-over Alignment* on page 108.

- `IlvTreeLayout.TIP_OVER_BOTH_SIDES`

  The children are arranged sequentially instead of in parallel. Whereas the alignment `TIP_OVER` arranges all children at the same side of the parent, this alignment arranges the children at both sides of the parent. For details see *Tip-over Alignment* on page 108.

- `IlvTreeLayout.MIXED`

  Each parent node can have a different alignment. The alignment of each individual node can be set with the result that different alignments can occur in the same graph.

To obtain the current setting, use the following method:

```
int getGlobalAlignment();
```

Alignment of Individual Nodes

All nodes have the same alignment unless the global alignment is set to `IlvTreeLayout.MIXED`. Only when the global alignment is set to mixed can each node have an individual alignment style.

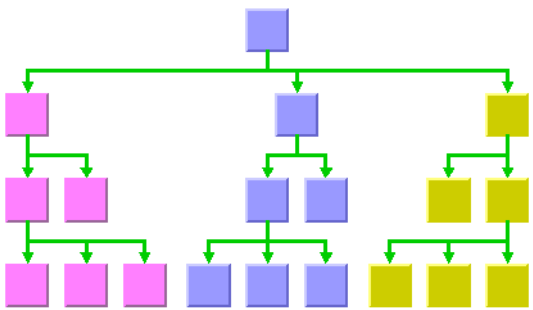


Figure 4.33 *Different Alignments Mixed in the Same Drawing*

To set and retrieve the alignment of an individual node, use the following methods:

```
void setAlignment(Object node, int alignment);

int getAlignment(Object node);
```

The valid values for `alignment` are:

- `IlvTreeLayout.CENTER` (the default)
- `IlvTreeLayout.BORDER_CENTER`
- `IlvTreeLayout.EAST`
- `IlvTreeLayout.WEST`
- `IlvTreeLayout.TIP_OVER`
- `IlvTreeLayout.TIP_OVER_BOTH_SIDES`

Tip-over Alignment

Normally, the children of a node are placed in a *parallel* arrangement with siblings directly neighbored to each other. Tip-over alignment means a *sequential* arrangement of the children instead.

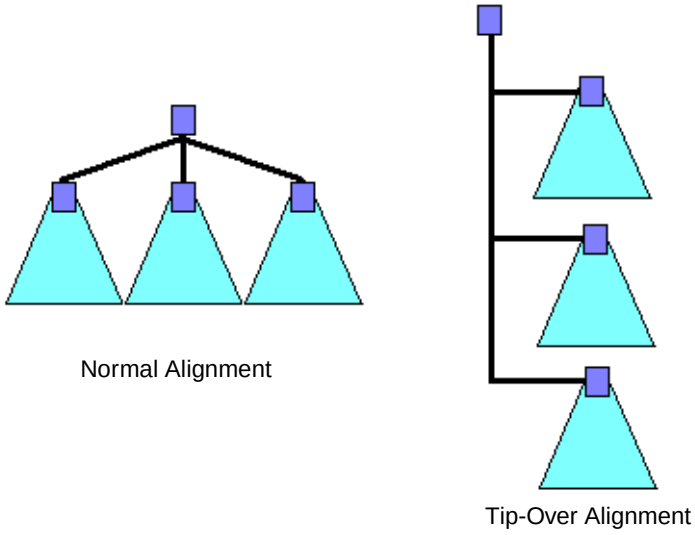


Figure 4.34 *Normal Alignment and Tip-over Alignment*

Tip-over alignment is useful when the tree has many leaves. With normal alignment, a tree with many leaves would result in the layout being very wide. If the global alignment style is set to tip-over, the drawing is very tall rather than wide. To balance the width and height of the drawing, you can set the global alignment to mixed:

```
layout.setGlobalAlignment(IlvTreeLayout.MIXED);
```

Also, you can set the individual alignment to tip-over for some parents with a high number of children as follows:

```
layout.setAlignment(parent, IlvTreeLayout.TIP_OVER);
```

Tip-over alignment can be specified explicitly for some or all of the nodes. Furthermore, the Tree Layout offers layout modes that automatically determine when to tip over, yielding a drawing that fits into a given aspect ratio. These layout modes are described in *Tip-Over Layout Modes* on page 126.

Besides the normal tip-over alignment, there is also a variant that distributes the subtrees on both sides of the center line that starts at the parent. You can specify this variant at a parent node with a high number of children as follows:

```
layout.setAlignment(parent, IlvTreeLayout.TIP_OVER_BOTH_SIDES);
```

The difference between normal tip-over alignment and tip-over at both sides is illustrated in Figure 4.35. Tip-over alignment works very well with the orthogonal link style (see *Link Style* on page 109).

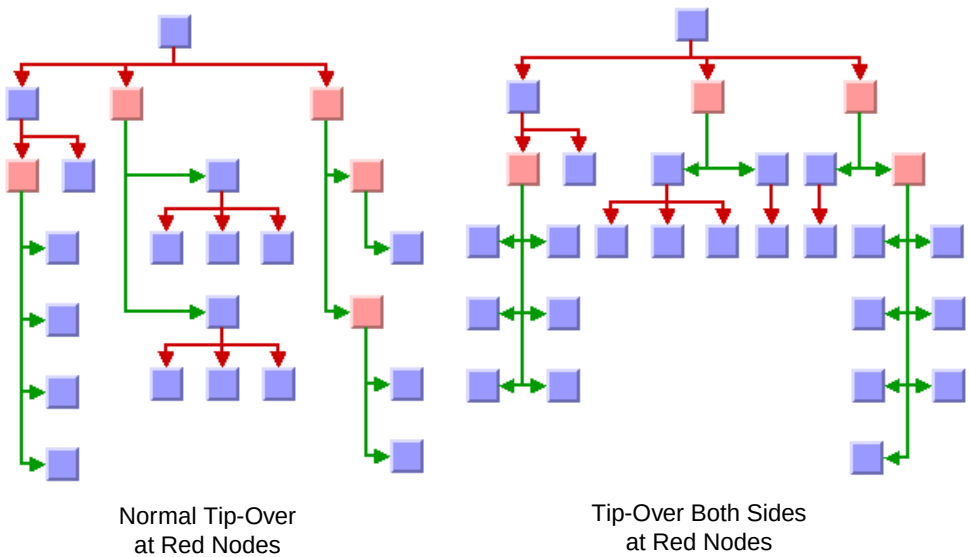


Figure 4.35 *Tip-over Alignment*

Link Style

When using the Tree layout, it is preferable to use link connectors of type `IlvRelativeLinkConnector` and links of type `IlvPolylineLinkImage` or `IlvSplineLinkImage`. The links can be straight or have a specific shape with intermediate points. You can specify that the links be reshaped into an “orthogonal” form. You can set the

link style globally, in which case all links have the same kind of shape, or locally on each link, in which case different link shapes occur in the same drawing.

> ***Note:*** *The layout algorithm may raise an* `InappropriateLinkException` *if layout is performed on an* `IlvGrapher`*, but links are neither a subclass of* `IlvPolylineLinkImage` *nor of* `IlvSplineLinkImage`*, or if connectors are not a subclass of* `IlvRelativeLinkConnector`*. You can use the methods* `EnsureAppropriateLinkTypes`, `EnsureAppropriateLinkConnectors` *or* `EnsureAppropriateLinks` *defined in* `IlvGraphLayoutUtil` *to automatically replace inappropriate links or link connectors before layout or when the* `IlvInappropriateLinkException` *is caught. For details on these methods, see the ILOG JViews Graph Layout Reference Manual. For details on the graph model, see Using the Graph Model on page 297.*

Global Link Style

To set the global link style, the following method is provided:

```
void setGlobalLinkStyle(int style);
```

The valid values for `style` are:

- `IlvTreeLayout.NO_RESHAPE_STYLE`

  None of the links is reshaped in any manner.

- `IlvTreeLayout.STRAIGHT_LINE_STYLE`

  All the intermediate points of the links (if any) are removed. This is the default value. See Figure 4.27 on page 94 and Figure 4.29 on page 95 as examples.

- `IlvTreeLayout.ORTHOGONAL_STYLE`

  The links are reshaped in an orthogonal form (alternating horizontal and vertical segments). See Figure 4.28 on page 94 and Figure 4.35 on page 109 as examples.

- `IlvTreeLayout.MIXED_STYLE`

  Each link can have a different link style. The style of each individual link can be set to have different link shapes occurring on the same graph.

To obtain the current choice, use the following method:

```
int getGlobalLinkStyle();
```

Individual Link Style

All links have the same style of shape unless the global link style is `IlvTreeLayout.MIXED_STYLE`. Only when the global link style is set to mixed can each link have an individual link style.

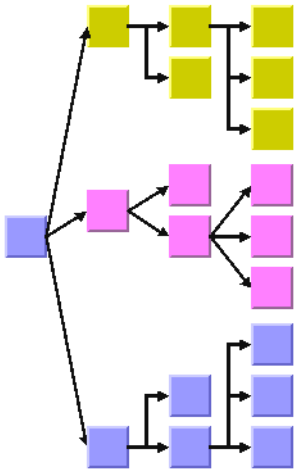


Figure 4.36 *Different Link Styles Mixed in the Same Drawing*

To set and retrieve the style of an individual link, use the following methods:

```
void setLinkStyle(Object link, int style);

int getLinkStyle(Object link);
```

The valid values for `style` are:

- `IlvTreeLayout.STRAIGHT_LINE_STYLE` (the default)
- `IlvTreeLayout.NO_RESHAPE_STYLE`
- `IlvTreeLayout.ORTHOGONAL_STYLE`

> ***Note:*** *The link style of a Tree Layout graph requires links in an* `IlvGrapher` *that can be reshaped. Links of type* `IlvLinkImage`, `IlvOneLinkImage`, `IlvDoubleLinkImage`, `IlvOneSplineLinkImage`, *and* `IlvDoubleSplineLinkImage` *cannot be reshaped. You should use the class* `IlvPolylineLinkImage` *or* `IlvSplineLinkImage` *instead.*

Connector Style

The layout algorithm automatically positions the end points of links (the connector pins) at the nodes. The connector style parameter specifies how these end points are calculated for the outgoing links at the parent node.

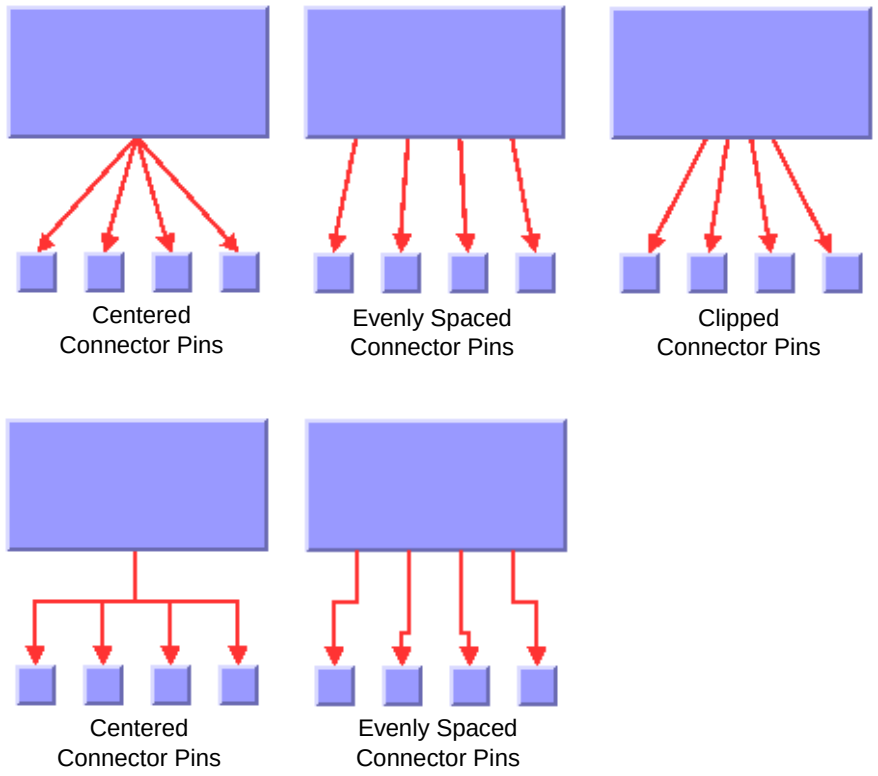


Figure 4.37 *Connector Styles*

To set the connector style, use the following method:

```
void setConnectorStyle(int style);
```

The valid values for `style` are:

- `IlvTreeLayout.CENTERED_PINS`

  The end points of the links are placed in the center of the border where the links are attached.

- `IlvTreeLayout.CLIPPED_PINS`

  Each link pointing to the center of the node is clipped at the node border. The connector pins are placed at the points on the border where the links are clipped. This style affects straight links. It behaves like centered connector pins for orthogonal links.

- `IlvTreeLayout.EVENLY_SPACED_PINS`

  The connector pins are evenly distributed along the node border. This style works for straight and orthogonal links.

- `IlvTreeLayout.AUTOMATIC_PINS`

  The connector style is selected automatically depending on the link style and the layout mode. In the nonradial modes, the algorithm always chooses centered pins. In the radial layout modes, it chooses clipped pins.

To obtain the current choice, use the following method:

```
int getConnectorStyle()
```

> ***Note:*** *The connector style parameter requires link connectors at the nodes of an* `IlvGrapher` *that allow connector pins to be placed freely at the node border. It is recommended that you use* `IlvRelativeLinkConnector` *for link connectors to be used in combination with* `IlvGrapher` *objects.*

Using a Link Connection Box Interface

By default, the connector style determines how the connection points of the links are distributed on the border of the bounding box of the nodes, symmetrically with respect to the middle of each side. Sometimes it may be necessary to place the connection points on a rectangle smaller or larger than the bounding box, eventually in a nonsymmetric way. For instance, this can happen when labels are displayed below or above nodes.

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkConnectionBoxInterface`. An example for the implementation of a link connection box interface is in *Link Connection Box* on page 29. To set a link connection box interface, call:

```
void setLinkConnectionBoxInterface(IlvLinkConnectionBoxInterface
interface)
```

The link connection box interface provides each node with a link connection box and a tangential shift offset that defines how much the connection points are “shifted” tangentially depending on which side the links connect.

Figure 4.38 illustrates the effects of customizing the connection box when the connector style is evenly spaced.

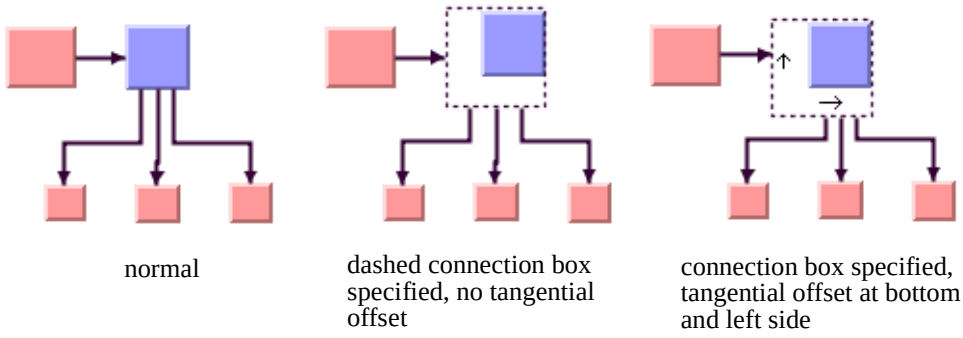

normal
dashed connection box specified, no tangential offset
connection box specified, tangential offset at bottom and left side

Figure 4.38 *Effect of Connection Box Interface*

On the left is the result without any connection box interface. The middle picture shows the effect if the connection box interface returns the dashed rectangle for the blue node but the tangential offset at all sides of the node is 0. Notice that the outgoing links are spaced according to the dashed rectangle, which appears too wide for the blue node in this situation. The picture on the right shows the effect of the connection box interface if, in addition, a positive tangential offset was specified for the bottom side and a negative offset was specified for the left side of the blue node.

Using a Link Clipping Interface

By default, the Tree Layout places the connection points of links at the border of the bounding box of the nodes. If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape. An example is shown in Figure 4.39.

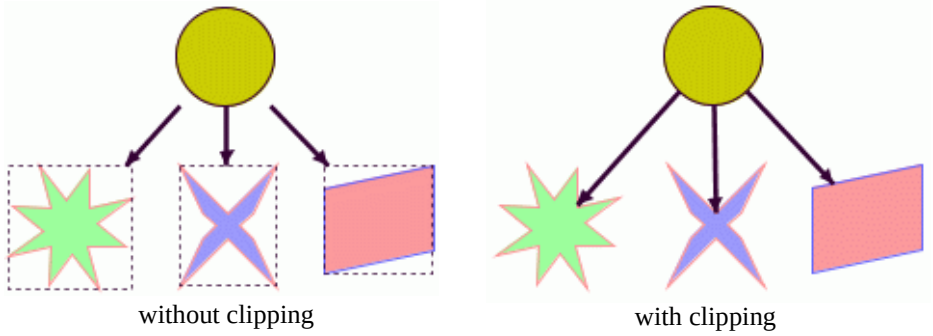


Figure 4.39 *Effect of Link Clipping Interface*

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkClipInterface`. An example for the implementation of a link clip interface is in *Link Clipping* on page 27. To set a link clip interface, call:

```
void setLinkClipInterface(IlvLinkClipInterface interface)
```

> ***Note:*** *In additionally to the link clip interface, the* `IlvClipLinkConnector` *can be used. This special link connector updates the clipped connection points automatically during interactive node movements.*

The connector style, the link connection box interface, and the link clip interface work together in the following way: by respecting the connector style, the proposed connection points are calculated on the rectangle obtained from the link connection box interface (or on the bounding box of the node, if no link connection box interface was specified). Then, the proposed connection point is passed to the link clip interface and the returned connection points are used to connect the link to the node.

An example of the combined effect is shown in Figure 4.40.

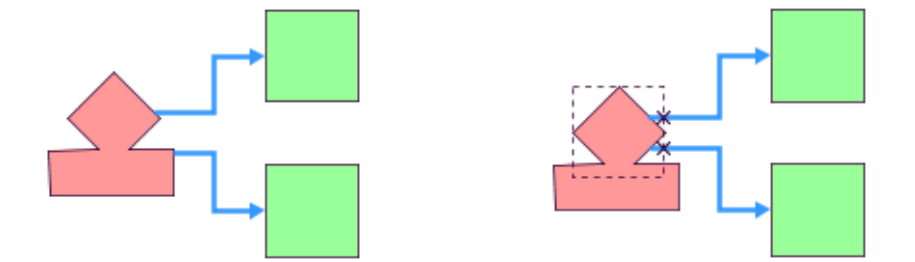


Figure 4.40 *Combined Effect of Link Clipping Interface and Link Connection Box*

If the links are clipped at the red node in Figure 4.40 (left), they appear unsymmetric with respect to the node shape, because the relevant part of the node (here: the upper rhombus) is not in the center of the bounding box of the node, but the proposed connection points are calculated with respect to the bounding box. This can be corrected by using a link connection box interface to explicitly specify a smaller connection box for the relevant part of the node (Figure 4.40 right) such that the proposed connection points are placed symmetrically at the upper rhombus of the node.

Spacing Parameters

The spacing of the layout is controlled mainly by three spacing parameters: the distance between a parent and its children, the minimal distance between siblings, and the minimal distance between nodes of different branches. For instance, if the flow direction is to the top

or bottom, the offset between parent and children is vertical, while the sibling offset and the branch offset are horizontal.

For tip-over alignment, an additional spacing parameter is provided: the minimal distance between branches starting at a node with tip-over alignment. This offset is always orthogonal to the normal branch offset. If the flow direction is to the top or bottom, the tip-over branch offset is vertical.

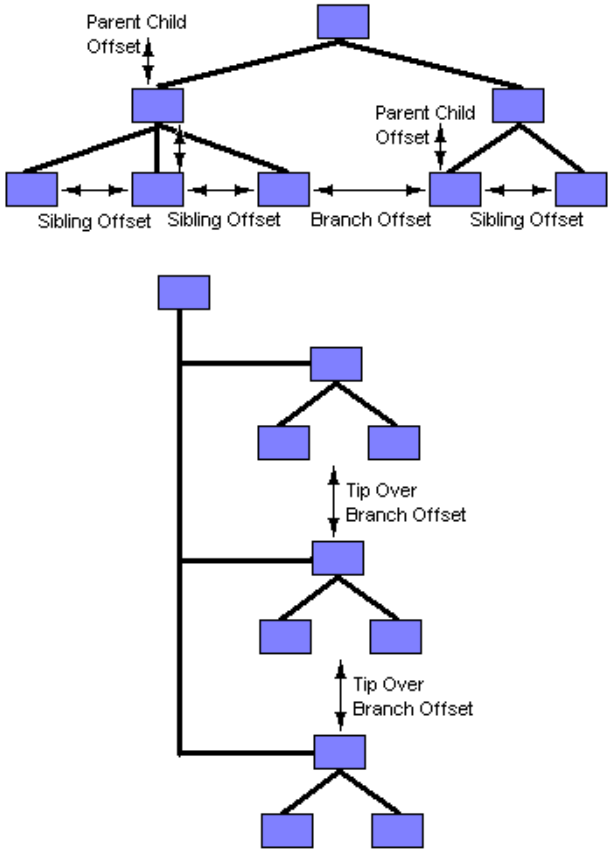


Figure 4.41 *Spacing Parameters*

The spacing parameters can be set by the following methods:

```
void setParentChildOffset(float offset);
void setSiblingOffset(float offset);
void setBranchOffset(float offset);
```

```
void setTipOverBranchOffset(float offset);
```

The current settings can be obtained by the corresponding methods:

```
float getParentChildOffset();

float getSiblingOffset();

float getBranchOffset();

float getTipOverBranchOffset();
```

For Experts: Additional Spacing Parameters

The spacing parameters normally specify the minimal offsets between the node borders. Hence, the layout algorithm places the nodes such that they do not overlap. You can also specify that the layout should ignore the node sizes, by setting:

```
layout.setRespectNodeSizes(false);
```

In this case, the spacing parameters are interpreted as the minimum distances between the node centers, and the node sides are not taken into account during the layout. However, if the specified offset parameters are now smaller than the node size, the nodes and links will overlap. This often happens with orthogonal links in particular. It makes sense to use this option only if all nodes have approximately the same size, all links are straight, and the spacing parameters are larger than the largest node.

If the link style is orthogonal, the shape of the links from the parent to its children looks like a fork (see Figure 4.33 on page 107). The position of the bend points in this shape can be influenced by the *orthogonal fork percentage*, a value between `0` and `100`. This is a percentage of the parent child offset. If the orthogonal fork percentage is `0`, the link shape forks directly at the parent node. If the percentage is `100`, the link shape forks at the child node. A good choice is between `25` and `75`. This percentage can be set and obtained by the following methods:

```
void setOrthForkPercentage(float percentage);

float getOrthForkPercentage();
```

If the link style is not orthogonal, links may overlap neighboring nodes. This happens only in a very few cases, for instance, when a link starts at a very small node that is neighbored by a very large node. This deficiency can be fixed by increasing the branch offset. However, this influences the layout globally, affecting nodes without that deficiency. To avoid a global change, you can change the *overlap percentage* instead, which is a value between 0 and 100. This value is used by an internal heuristic of the layout algorithm that considers a node to be smaller by this percentage. The default percentage is `30`. This usually results in better usage of the space. However, if very small nodes are neighbored to very large nodes, it is recommended to decrease the overlap percentage or to set it to `0` to switch this heuristic off to avoid links overlapping nodes. The overlap percentage can be set and obtained by the following methods:

```
void setOverlapPercentage(float percentage);
float getOverlapPercentage();
```

> ***Note:*** *It is recommended that you always set the orthogonal fork percentage to a value larger than the value of the overlap percentage.*

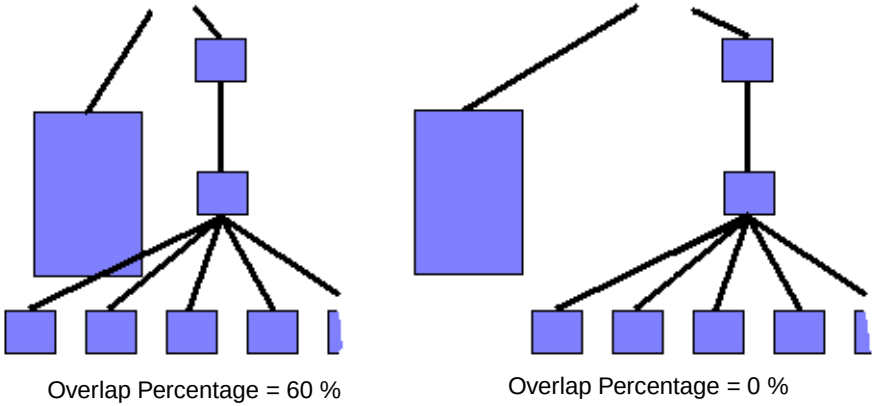


Figure 4.42 *Effect of Using the Overlap Percentage*

Level Layout Mode

The level layout mode partitions the node into levels and arranges the levels horizontally or vertically. The root is placed at level 0, its children at level 1, the children of those children at level 2, and so on. In contrast to the free layout mode, in level layout mode the nodes of the same level are justified with each other even if they are not siblings (that is, they do not have the same parent). Figure 4.43 shows the same graph in free layout mode and in level layout mode.

The following statement sets the level layout mode:

```
layout.setLayoutMode(IlvTreeLayout.LEVEL);
```

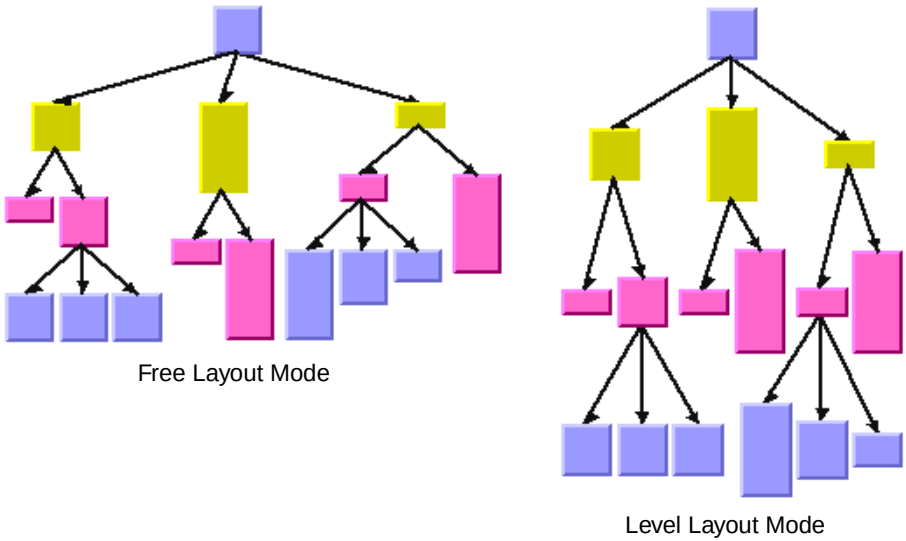


Figure 4.43 *Free Layout Mode and Level Layout Mode*

General Parameters

Most layout parameters that work for the free layout mode work as well for the level layout mode. You can set the flow direction, the spacing offsets, the global or individual link style, and the global or individual alignment. See *Free Layout Mode* on page 104 for details.

The differences from the free layout mode are:

- The tip-over alignment does not work in level layout mode.
- The parent-child offset parameter controls the spacing between the levels. In level layout mode, it is the *minimal* distance between parent and its children, while in free layout mode, it is the *exact* distance between parent and its children.
- The overlap percentage has no effect in level layout mode.

Level Alignment

In level layout mode with flow direction to the top or bottom, the nodes are organized in horizontal levels such that the nodes of the same level are placed approximately at the same y-coordinate. The nodes can be justified, depending on whether the top border, the bottom border, or the center of all nodes of the same level should have the same y-coordinate.

In flow direction to the left or right, the nodes are organized in vertical levels approximately at the same x-coordinate. The nodes of the same level can be justified at the left border, at the right border, or at the center.

To distinguish the level alignment independently from the flow direction, we use the directions north and south (see *Using Compass Directions for Situating Layout Parameters* on page 103). The north border of a node is the border that is closer to the level where its parent is placed, and the south border of a node is the border that is closer to the level where its children are placed. If the flow direction is to the bottom, the level alignment north means that the nodes are justified at the top border, and south means that the nodes are justified at the bottom border. If the flow direction is to the top, it is converse: north means the bottom border and south means the top border. If the flow direction is to the right, then north means the left border and south means the right border.

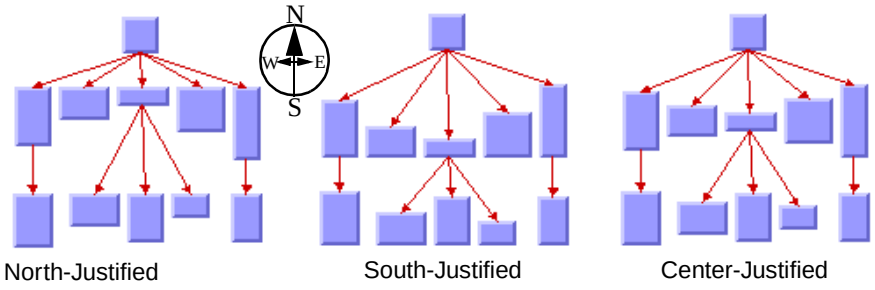


Figure 4.44 *Level Alignment*

To specify the level alignment, use the following method:

```
void setLevelAlignment(int alignment);
```

To obtain the current value, use the method:

```
int getLevelAlignment();
```

The valid values for `alignment` are:

- `IlvTreeLayout.CENTER` (the default)
- `IlvTreeLayout.NORTH`
- `IlvTreeLayout.SOUTH`

Radial Layout Mode

The radial layout mode partitions the node into levels and arranges the levels in circles around the root node. Figure 4.45 shows an example of the radial layout mode. The compass icons show the compass directions in this drawing.

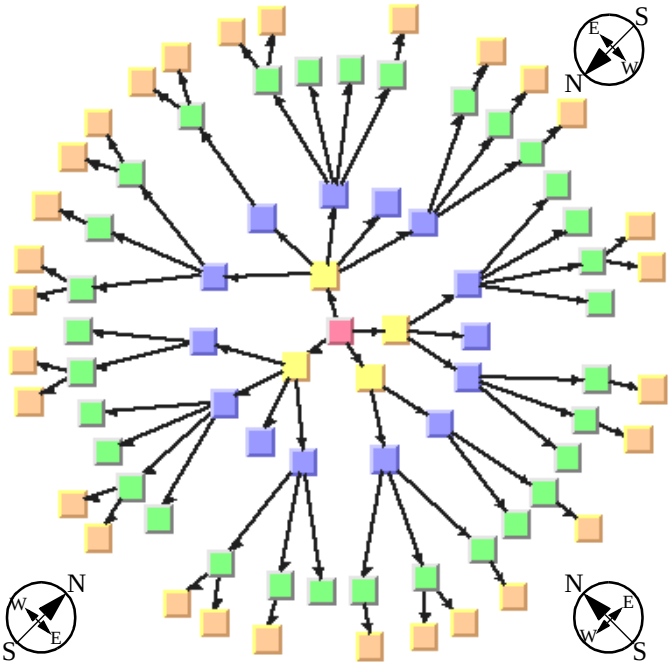


Figure 4.45 *Radial Layout Mode*

The following statement sets the radial layout mode:

```
layout.setLayoutMode(IlvTreeLayout.RADIAL);
```

General Parameters

Most layout parameters that work for the free and level layout mode work as well for the radial layout mode. You can set the spacing offsets, the level alignment, the global or individual link style, and the global or individual alignment. See *Free Layout Mode* on page 104 and *Level Layout Mode* on page 118 for details.

The radial layout mode differs from the other layout modes as follows:

- The tip-over alignment does not work in radial layout mode.
- The orthogonal link style does not work in radial layout mode.
- The clipped connector style is always used.
- The parent-child offset parameter controls the minimal distance between the circular levels. However, it is sometimes necessary to increase the offset between circular levels to obtain enough space on the circle to place all nodes of a level.

- The level alignment *north* indicates alignment at the inner border of the circular level (that is, towards the root), and the level alignment *south* indicates alignment at the outer border of the circular level (that is, away from the root).
- The level alignments *north* and *south* sometimes result in overlapping nodes.
- The overlap percentage has no effect in radial layout mode.

Alternating Radial Mode

If levels of the graph contain many nodes, it is sometimes necessary to increase the radius of the circular level to provide enough space on the circumference of the circle for all the nodes. This may result in a considerable distance from the previous level. To avoid this, there is an *alternating* radial mode. The alternating radial mode places the nodes of a level alternating between two circles instead of one circle, resulting better usage of the space of the layout.

The alternating radial mode uses two circles only when necessary. For many small and light trees, there will be no difference from the normal radial mode. Only for large graphs with a large number of children will the alternating radial mode have an effect.

The following statement sets the alternating radial layout mode:

```
layout.setLayoutMode(IlvTreeLayout.ALTERNATING_RADIAL);
```

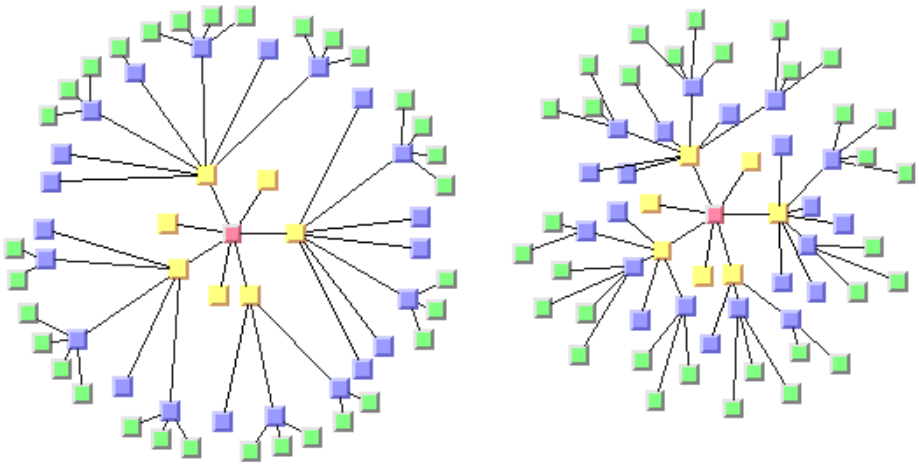


Figure 4.46 *Radial Layout Mode (right) and Alternating Radial Layout Mode (left)*

Aspect Ratio

If the drawing area is not a square, arranging the levels as circles is not always the best choice. You can specify the aspect ratio of the drawing area to better fit the layout to the drawing area. In this case, the algorithm uses ellipses instead of circles. See Figure 4.29 on page 95 for an example.

If the drawing area is a view (a subclass of `IlvManagerView`), you can use the following method:

```
void setAspectRatio(IlvManagerView view);
```

If the drawing area is given only as a rectangle, use the following:

```
void setAspectRatio(IlvRect rect);
```

If neither a view nor a rectangle is specified, you can calculate the aspect ratio from the width and height of the drawing area as `aspectRatio = width/height` and use the following method:

```
void setAspectRatio(float aspectRatio);
```

To obtain the current aspect ratio, call:

```
float getAspectRatio();
```

Spacing Parameters

The spacing parameters of the radial layout mode are controlled by the same methods as used for the free and level layout modes:

```
void setParentChildOffset(float offset);
```

```
void setSiblingOffset(float offset);
```

```
void setBranchOffset(float offset);
```

Note that the sibling and branch offsets are minimal distances tangential to the circles or ellipses, while the parent-child offset is a minimal distance radial to the circles or ellipses. Figure 4.47 shows the spacing parameters in radial layout mode.

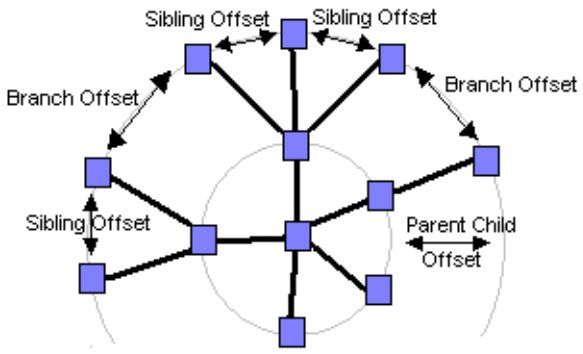


Figure 4.47 *Spacing Parameters in Radial Layout Mode*

For Experts: Adding an Invisible Root to the Layout

If the graph contains several trees that are disconnected from each other, the layout places them individually next to each other. Each connected component has its own radial structure with circular layers. However, sometimes it is appropriate to fit all connected components into a single circular layer structure. Conceptually, this is done by adding an invisible root at the center and connecting all disconnected trees to this root. Figure 4.48 shows the effect of using an invisible root. This works only if the generic mechanism to lay out connected components is switched off using the following method:

```
layout.setLayoutOfConnectedComponentsEnabled(false);
```

To use an invisible root to combine disconnected components, use the following code:

```
layout.setInvisibleRootUsed(true);
```

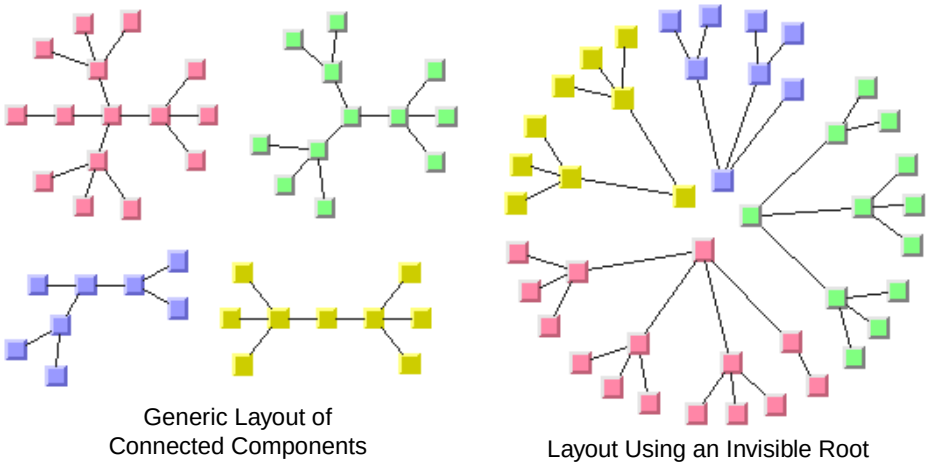


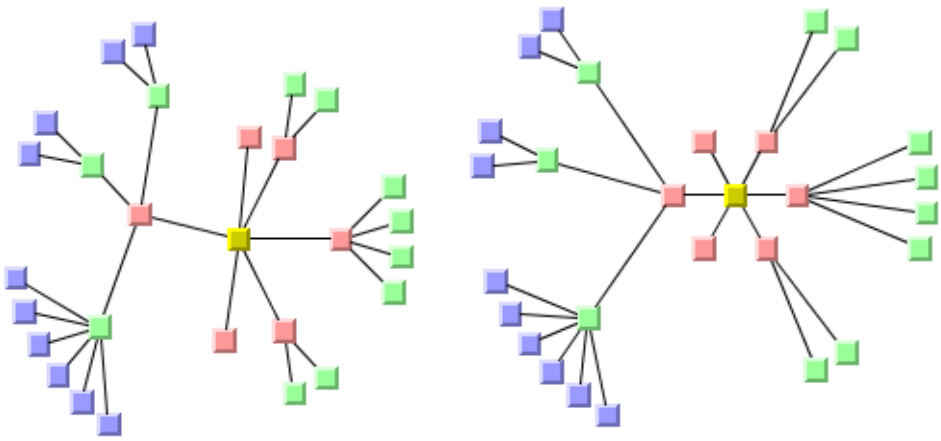
Figure 4.48 *Layout of Connected Components without and with an Invisible Root*

For Experts: Even Spacing for the First Circle

The radial mode is designed to optimize the space such that the circles have a small radius and the overall space for the entire layout is small. To achieve this, the layout algorithm may create larger gaps on the inner circles for better space usage of the outer circles. This may produce unevenly spaced circles, most notably for the first circle where all nodes have the same parent node.

To avoid this effect, you can force the nodes to be evenly spaced on the entire first circle. Depending on the structure of the graph, this may cause the overall layout to waste more space on the other circles but may produce a more pleasing graph. To enable this feature, call:

```
layout.setFirstCircleEvenlySpacing(true);
```



Unevenly spaced first (red) circle

Evenly spaced first (red) circle

Figure 4.49 *Evenly and Unevenly Spaced First Circle*

For Experts: Setting a Maximal Children Angle

If a node has a lot of children, they may extend over a major portion of the circle and, therefore, are placed nearly 360 degrees around the node. This can result in links overlapping some nodes. The deficiency can be fixed by increasing the offset between parent and children. However, this affects the layout globally which means that nodes without the deficiency are also affected.To avoid a global change such as this, you can limit the maximal angle between the two rays from the parent (if it is not the root) to its two outermost children. This increases the offset between parent and children only where necessary.

In Figure 4.50, you can see in the layout on the left that many of the links overlap other nodes. In the layout on the right, you can see how this problem was solved by setting a maximal children angle between two rays from a parent to the two outermost children.

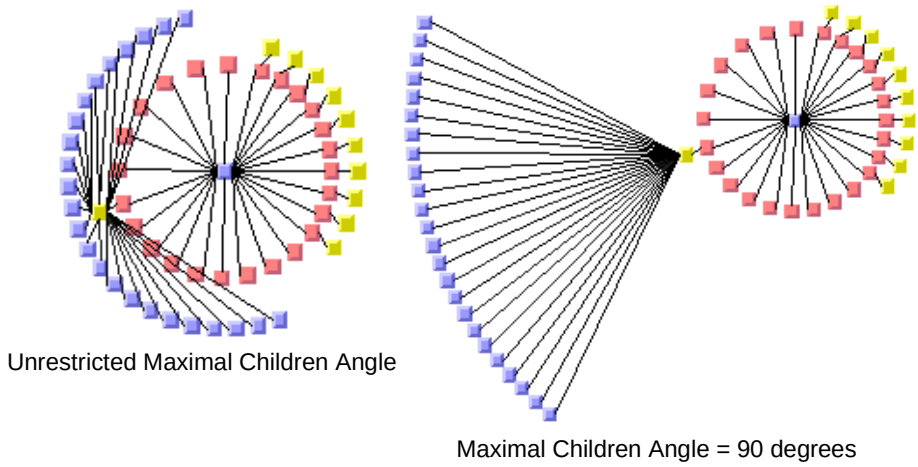


Figure 4.50 *Maximal Children Angle*

Use the following method to set and obtain the angle in degrees:

```
void setMaxChildrenAngle(int angle);

int getMaxChildrenAngle();
```

Recommended values are between `30` and `180`. Setting the value to `0` means the angle is unrestricted. The calculation of the angle is not very precise above 180 degrees or if the aspect ratio is not 1.0.

Tip-Over Layout Modes

As in radial layout mode, drawing in free layout mode can be adjusted to the aspect ratio of the drawing area. Free layout mode can also use tip-over alignment to balance the drawing between the height and depth.

While tip-over alignment can be specified explicitly for individual nodes, the Tree Layout algorithm also has layout modes that automatically use tip-over alignment when needed. These are the tip-over layout modes.

The tip-over layout modes work as follows: Several trial layouts are performed in free layout mode. For each trial, certain tip-over alignments are set for individual nodes, while the specified alignment of all other nodes is preserved. The algorithm picks the trial layout that best fits the specified aspect ratio of the drawing area.

The aspect ratio can be set by one of the following methods (see *Aspect Ratio* on page 122 in the Radial Layout Mode):

```
void setAspectRatio(IlvManagerView view);
```

```
void setAspectRatio(IlvRect rect);

void setAspectRatio(float aspectRatio);
```

The tip-over modes are slightly more time-consuming than the other layout modes. For very large trees, it is recommended that you set the allowed layout time to a high value (for instance, 60 seconds) when using the tip-over modes. Use the following call:

```
layout.setAllowedTime(60000);
```

By using this call, you avoid running short of time for sufficient iterations of the layout algorithm. Because it would be too time-consuming to check all possibilities of tip-over alignment use, there are heuristics that check only certain trials according to the following different strategies, illustrated in Figure 4.51:

- *Tip Leaves Over*
- *Tip Roots Over*
- *Tip Roots and Leaves Over*
- *Tip Over Fast*

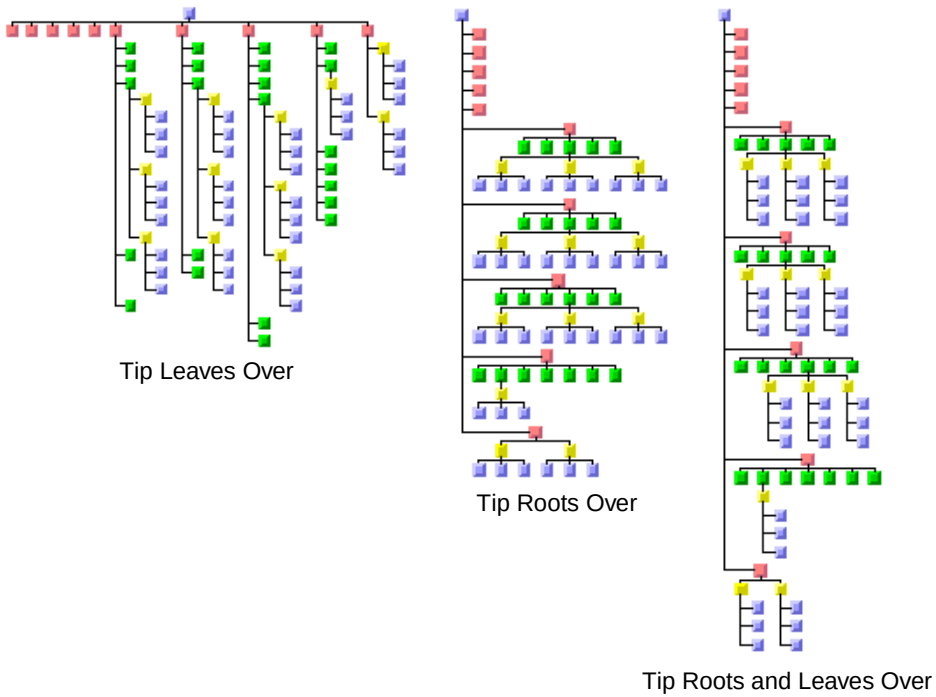


Figure 4.51 *Tip-Over Strategies*

Tip Leaves Over

To use this tip-over strategy, set the layout mode as follows:

```
layout.setLayoutMode(IlvTreeLayout.TIP_LEAVES_OVER);
```

The heuristic first tries the layout without any additional tip-over options. Then it tries to tip over the leaves, then the leaves and their parents, then additionally the parents of these parents, and so on. As a result, the nodes closest to the root use normal alignment and the nodes closest to the leaves use tip-over alignment.

Tip Roots Over

To use this tip-over strategy, set the layout mode as follows:

```
layout.setLayoutMode(IlvTreeLayout.TIP_ROOTS_OVER);
```

The heuristic first tries the layout without any additional tip-over options. Then it tries to tip over the root node, then the root and its children, then additionally the children of these children, and so on. As a result, the nodes closer to the leaves use normal alignment and the nodes closer to the root use tip-over alignment.

Tip Roots and Leaves Over

To use this tip-over strategy, set the layout mode as follows:

```
layout.setLayoutMode(IlvTreeLayout.TIP_ROOTS_AND_LEAVES_OVER);
```

The heuristic first tries the layout without any additional tip-over options. Then it tries to tip over the root node and the leaves simultaneously; then the root and its children, and the leaves and its parent; then additionally the children of these children and the parents of these parents, and so on. As result, the nodes in the middle of the tree use normal alignment and the nodes closest to the root or leaves use the tip-over alignment.

This is the slowest strategy because it includes all trials of the strategy "*tip leaves over*" as well as all tries of the strategy "*tip roots over.*"

Tip Over Fast

The fast tip-over provides a compromise among all other strategies. The heuristic tries a small selection of the other strategies, not all possibilities. Therefore, it is the fastest strategy for large graphs.

To use this strategy, set the layout mode as follows:

```
layout.setLayoutMode(IlvTreeLayout.TIP_OVER);
```

It is possible that all four strategies yield the same result because the strategies are not disjoint; that is, certain trials are performed in all four strategies. In addition, the tip-over modes do not necessarily produce the optimal layout that gives the best possible fit to the aspect ratio. The reason is that some unusual configurations of tip-over alignment are never tried because doing so would cause the running time to be too high.

For Experts: Additional Tips and Tricks

Specifying East-West Neighbors

You can specify that two unrelated nodes must be directly neighbored in a direction perpendicular to the flow direction. In the level and radial layout modes, both nodes are placed in the same level next to each other. In the free layout and tip-over modes, both nodes are placed aligned at the north border. Such nodes are called *east-west neighbors* because one node is placed as the direct neighbor on the east side of the other node. The other node becomes the direct neighbor on the west side of the first node. (See also *Using Compass Directions for Situating Layout Parameters* on page 103).

Technically, both nodes are treated as parent and child, even if there may be no link in between. Therefore, one of the two nodes can have a real parent, but the other node should not because its virtual parent is its *east-west neighbor.*

The east-west neighbor feature can be used, for example, for annotating nodes in a typed syntax tree occurring in compiler construction. Figure 4.52 shows an example of such a tree.

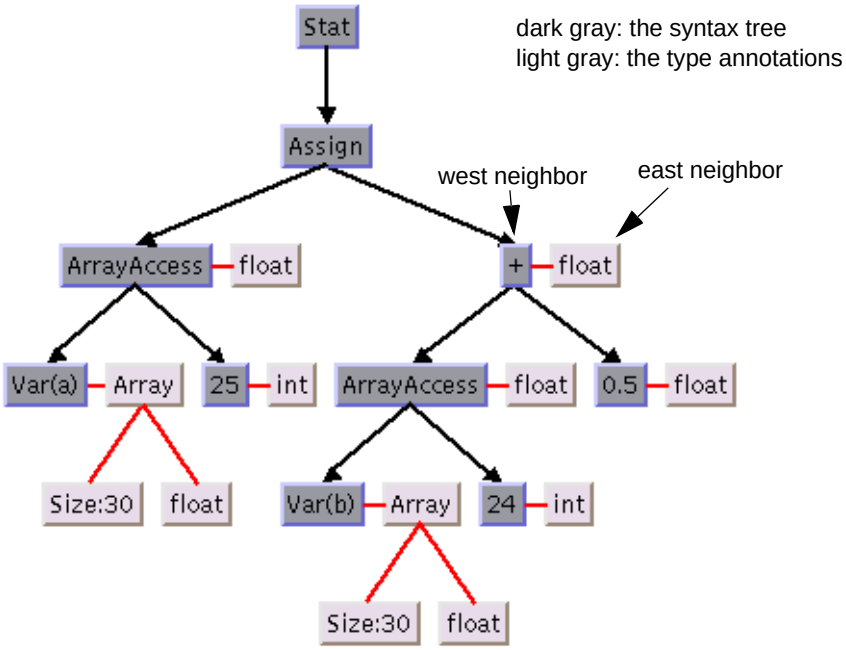


Figure 4.52 *Annotated Syntax Tree of Statement a[25] = b[24] + 0.5;*

To specify that two nodes are east-west neighbors, use the following method:

```
void setEastWestNeighboring(Object eastNode, Object westNode);
```

You can also use the following method, which is identical except for the reversed parameter order:

```
void setWestEastNeighboring(Object westNode, Object eastNode);
```

If the flow direction is to the bottom, the latter method may be easier to remember because, in this case, west is to the left of east in the layout, which is similar to the text flow of the parameters.

To obtain the node that is the east or west neighbor of a node, use the following calls:

```
Object getEastNeighbor(Object node);
```

```
Object getWestNeighbor(Object node);
```

Note that each node can have maximally one east neighbor and one west neighbor because they are *directly* neighbored. If more than one direct neighbor is specified, it is partially ignored. Cyclic specifications can cause conflict as well. For instance, if node B is the east neighbor of node A and node C is the east neighbor of B, then node A cannot be the east neighbor of C. (Strictly speaking, such cycles could be technically possible in some situations in the radial layout mode, but nonetheless they are not allowed in any layout mode.)

If B is the east neighbor of A, then A is automatically the west neighbor of B. On the other hand, the east neighbor of A can itself have another east neighbor. This allows creating chains of east-west neighbors, which is a common way to visualize lists of trees. Two examples are shown in Figure 4.53.

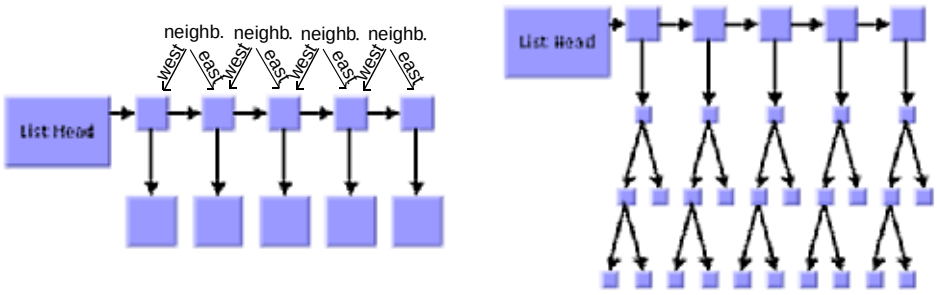


Figure 4.53 *Chains of East-West Neighbors to Visualize Lists of Trees*

Retrieving Link Categories

The Tree Layout algorithm works on a spanning tree, as mentioned in a *Brief Description of the Algorithm* on page 97. If the graph to be laid out is not a pure tree, the algorithm ignores

some links. To perform special treatment of such links, you can obtain a list of non-tree links.

Because there are parents and children in the spanning tree, we distinguish the following link categories:

- A forward tree link is a link from a parent to its child.
- A backward tree link is a link from a child to its parent. If the link is drawn as a directed arrow, the arrow will point in the opposite direction to the flow direction.
- A non-tree link is a link between two unrelated nodes; neither one is a child of the other.

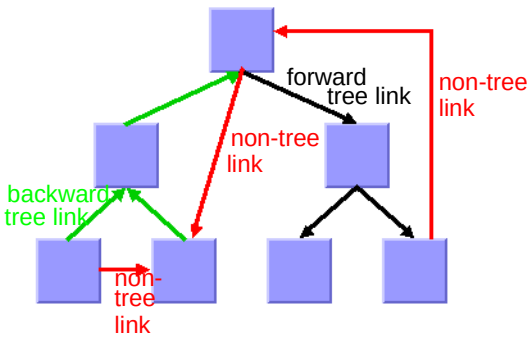


Figure 4.54 *Link Categories*

The layout algorithm uses these link categories internally but does not store them permanently for the sake of conserving time and memory efficiency. If you want to perform special treatment on some link categories (for example, to call the Link Layout on the non-tree links), you must specify *before the layout* that you want to access the link categories *after the layout*. In order to do this, use the method `setCategorizingLinks` in the following way:

```
layout.setCategorizingLinks(true);
// now perform a layout
layout.performLayout();
// now you can access the link categories
```

After the layout, the link categories can be obtained by the following methods:

```
Enumeration getCalcForwardTreeLinks();

Enumeration getCalcBackwardTreeLinks();

Enumeration getCalcNonTreeLinks();
```

The link category data gets filled each time the layout is called, unless you set the method `setCategorizingLinks` back to `false`.

Sequences of Layouts with Incremental Changes

You can work with trees that have become out-of-date, for example, those that need to be extended with more children. If you perform a layout after an extension, you probably want to identify the parts that had already been laid out in the original graph. The Tree Layout algorithm supports these incremental changes in incremental mode because it takes the previous positions of the nodes into account. It preserves the relative order of the children in the subsequent layout.

In nonincremental mode, the Tree Layout algorithm calculates the order of the children from the node order given by the attached graph model (or grapher). In this case, the layout is independent from the positions of the nodes before layout. It does not preserve the relative order of the children in subsequent layouts.

The incremental mode is enabled by default.To disable the incremental mode, call:

```
layout.setIncrementalMode(false);
```

Interactive Editing

The fact that the relative order of the layout is preserved is particularly useful during interactive editing. It allows you to correct the layout easily. For instance, if the first layout places a node A left to its sibling node B but you need to reverse the order, you can simply move node A to the right of node B and start a new layout to clean up the drawing. In the second layout, A remains to the right of B, and the subtree of A will “follow” node A.

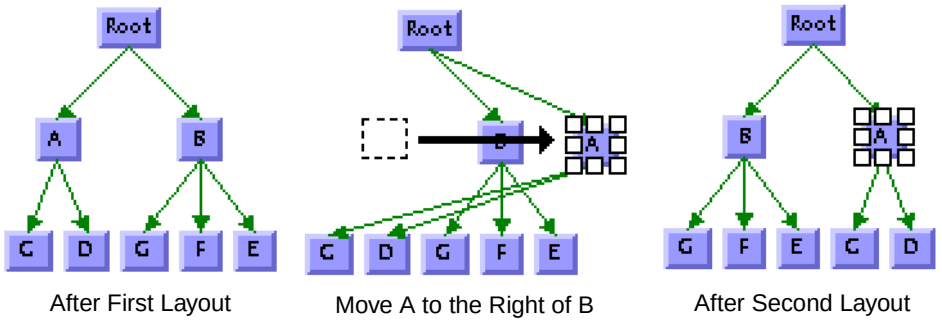


Figure 4.55 *Interactive Editing to Achieve a Specific Order of Children*

Specifying the Order of Children

Some applications require a specific relative order of the children in the tree. This means that, for instance, when the flow direction is to the bottom, which child must be placed to the left of another child. Even if the graph has never been laid out, you can use the coordinates to specify a certain order of the children at a node. You can use the following:

- First, make sure that the incremental mode is enabled.

- In free and level layout modes with flow direction to the bottom or top, determine the maximal width `W` of all nodes. Simply move the child that should be in the leftmost position to the coordinate `(0, 0)`, and the child that should get the *i*th relative position (in order from left to right) to the coordinate `((W+1)*i, 0)`.
- If the flow direction is to the left or to the right, determine the maximal height `H` of all nodes. Move the child that should be in the topmost position to the coordinate `(0, 0)` and the child that should get the *i*th relative position (in the order from top to bottom) to coordinate `(0, (H+1)*i)`.
- In the radial layout modes, determine the maximal diagonal `D = W2 + H` of all nodes. If the position of the parent is `(x, y)` before the layout, move the child that should be the first in the circular order to the coordinate `(x, y+D)` and the child that should get the *i*th relative position in the circular order to coordinate `(x+D*i, y+D)`.

  If you want to specify a relative order for all nodes in radial layout mode, you must do this for the parents before you do it for the children. In this case, moving the children can be performed easily during a depth-first traversal from the root to the leaves.

The layout that is performed after moving the children arranges the children with the desired relative order.

Hierarchical Layout

In this section, you will learn about the *Hierarchical Layout* algorithm from the ILOG JViews Graph Layout module (class `IlvHierarchicalLayout` from the package `ilog.views.graphlayout.hierarchical`).

Samples

Here are some sample drawings produced with the Hierarchical Layout:

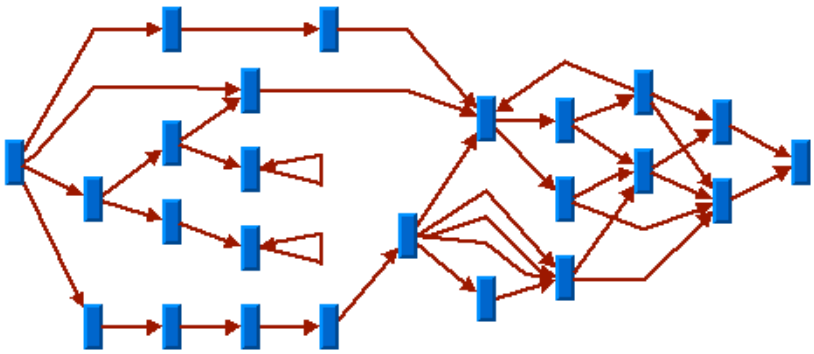


Figure 4.56 *Sample Layout with Self-Loops, Multiple Links, and Cycles*

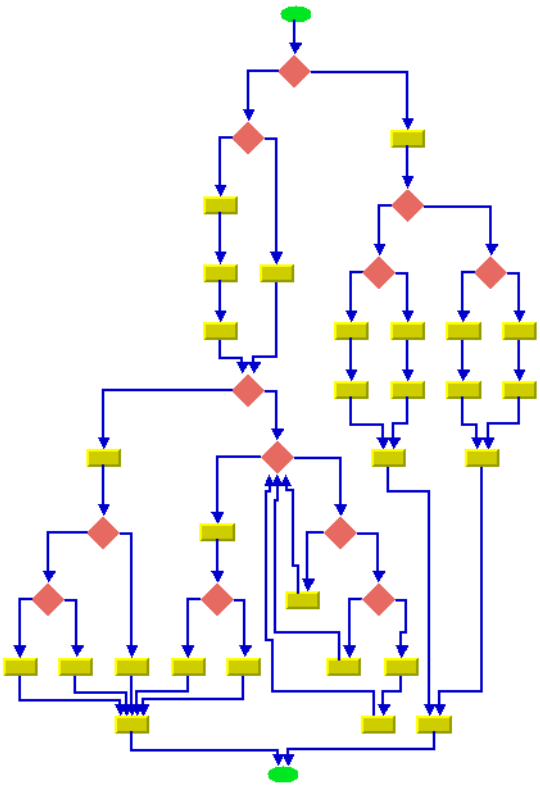


Figure 4.57 *Flowchart with Orthogonal Link Style*

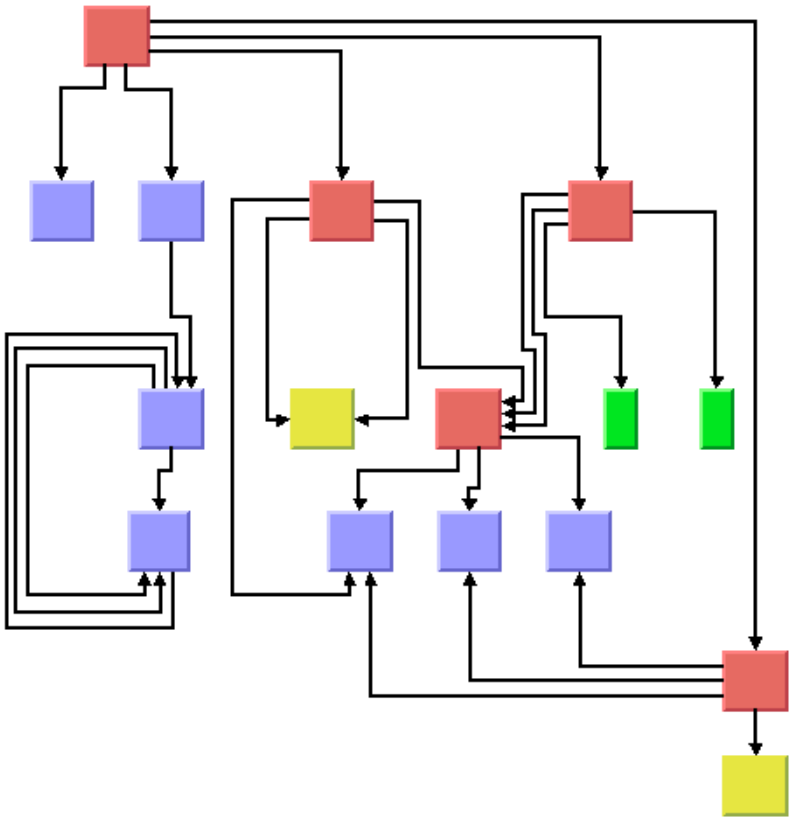


Figure 4.58 *Sample Layout with Ports and Orthogonal Link Style*

What Types of Graphs?

Any type of graph:

- Preferably graphs with directed links (The algorithm takes the link direction into account.)
- Connected and disconnected graphs
- Planar and nonplanar graphs

Application Domains

Application domains of the Hierarchical Layout include:

- Electrical engineering (logic diagrams, circuit block diagrams)

- Industrial engineering (industrial process diagrams, schematic design diagrams)
- Business processing (workflow diagrams, process flow diagrams, PERT charts)
- Software management/software (re-)engineering (UML diagrams, flowcharts, data inspector diagrams, call graphs)
- Database and knowledge engineering (database query graphs)
- CASE tools (designs diagrams)

Features

- Organizes nodes without overlaps in horizontal or vertical levels.
- Arranges the graph such that the majority of links are short and flow uniformly in the same direction (from left to right, from top to bottom, and so on).
- Reduces the number of link crossings. Most of the time, produces drawings with no crossings or only a small number of crossings.
- Often produces balanced drawings that emphasize the symmetries in the graph.
- Supports self-links (that is, links with the same origin and destination node), multiple links between the same pair of nodes, and cycles.
- Efficient, scalable algorithm. Produces a nice layout for most sparse and medium-dense graphs relatively quickly, even if the number of nodes is very large.
- Provides several alignment and offset options.
- Supports port specifications where links attach the nodes. Allows you to specify which side of a node (top, bottom, left, right) a link can be connected to or to specify which relative port position should be used for the connection.
- Supports layout constraints. Allows you to specify relative positional constraints, for instance, that a node is above another node or left of another node.
- Incremental and nonincremental mode. In incremental mode, the previous position of nodes are taken into account. Positions the nodes without changing the relative order of the nodes so that the layout is stable on incremental changes of the graph.
- The computation time depends on the number of nodes, the number of levels, and the number of links that cross several levels. Most of the time, the links are placed between adjacent levels, which keeps the computation time small.

Limitations

- The algorithm tries to minimize the number of link crossings (which is generally an NP-complete problem). It is mathematically impossible to solve this problem quickly for any graph size. Therefore, the algorithm uses a very fast heuristic that obtains a good layout, but not always with the theoretical minimum number of link crossings.
- The algorithm tries to place the nodes such that all links point uniformly in the same direction. It is impossible to place cycles of links in this way. For this reason, it sometimes produces a graph where a small number of links are reversed to point into the opposite direction. The algorithm tries to minimize the number of reversed links (which, again, is an NP-complete problem). Therefore, the algorithm uses a very fast heuristic resulting in a good layout, but not always with the theoretical minimum number of reversed links.
- The computation time required to obtain an appropriate drawing depends most significantly on the number of bends in the links. Since the algorithm places one bend whenever a link crosses a level, the number of bends can grow relatively quickly if the layout requires many long links that span several levels. Therefore, the layout process may become very time-consuming for dense graphs (the number of links is relatively high compared to the number of nodes) or for graphs that require a large number of node levels.

Brief Description of the Algorithm

This algorithm works in four steps.

Step 1: Leveling

The nodes are partitioned into groups. Each group of nodes forms a level. The objective is to group the nodes in such a way that the links always point from a level with smaller index to a level with larger index.

Step 2: Crossing Reduction

The nodes are sorted within each level. The algorithm tries to keep the number of link crossings small when, for each level, the nodes are placed in this order on a line (see Figure 4.59). This ordering results in the relative position index of each node within its level.

Step 3: Node Positioning

From the level indices and position indices, balanced coordinates for the nodes are calculated. For instance, for a layout where the link flow is from top to bottom, the nodes are placed along horizontal lines such that all nodes belonging to the same level have (approximately) the same y-coordinate. The nodes of a level with a smaller index have a smaller y-coordinate than the nodes of a level with a higher index. Within a level, the nodes with a smaller position index have a smaller x-coordinate than the nodes with a higher position index.

Step 4: Link Routing

The shapes of the links are calculated such that the links bypass the nodes at the level lines. In many cases, this requires that a bend point be created whenever a link needs to cross a level line. In a top-to-bottom layout, these bend points have the same y-coordinate as the level line they cross. (Note that these bend points also obtain a position index).

Figure 4.59 shows how the Hierarchical Layout algorithm uses the level and positions indices to draw the graph.

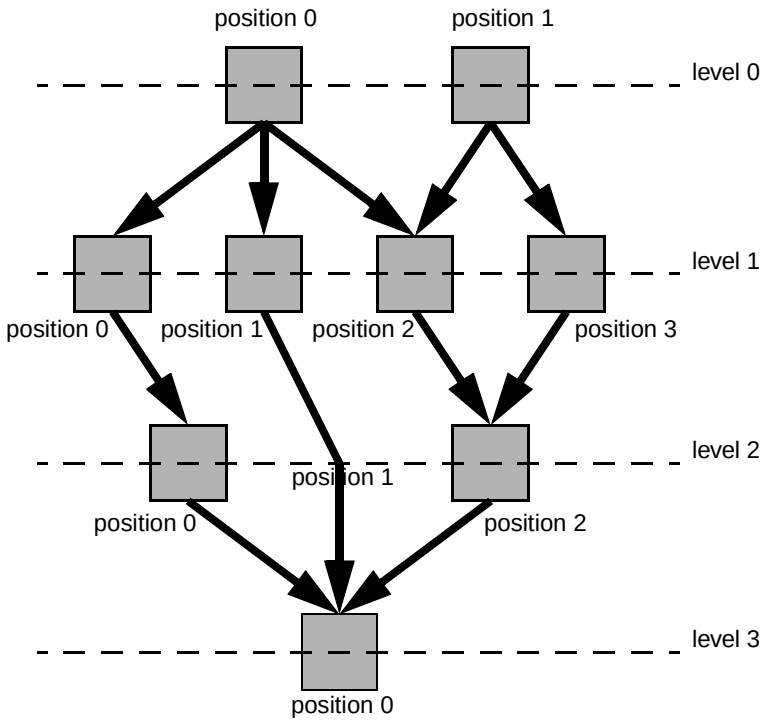


Figure 4.59 *Level and Position Indices*

The steps of the layout algorithm can be affected in several ways. For instance, you can specify the desired level index that the algorithm should choose for a node in Step 1 or the desired relative node position within the level in Step 2. You can also specify the justification of the nodes within a level and the style of the links shapes.

Code Sample

Below is a code sample using the `IlvHierarchicalLayout` class:

```
...
```

```
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.hierarchical.*;
 ...
IlvGrapher grapher = new IlvGrapher();
IlvManagerView view = new IlvManagerView(grapher);

 ... /* Fill in the grapher with nodes and links here */

IlvHierarchicalLayout layout = new IlvHierarchicalLayout();
layout.attach(grapher);
try {
        IlvGraphLayoutReport layoutReport = layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Generic Features and Parameters

The `IlvHierarchicalLayout` class supports the following generic features defined in the `IlvGraphLayout` class:

- *Allowed Time*
- *Layout of Connected Components*
- *Link Clipping*
- *Link Connection Box*
- *Percentage Completion Calculation*
- *Preserve Fixed Links*
- *Preserve Fixed Nodes*
- *Save Parameters to Named Properties*
- *Stop Immediately*

The following paragraphs describe the particular way in which these parameters are used by this subclass.

Allowed Time

The layout algorithm stops if the allowed time setting has elapsed. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Allowed Time* on page 22.) If the layout stops early because the allowed time has elapsed, the nodes and links are not moved from

their positions before the layout call and the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Layout of Connected Components

The layout algorithm can utilize the generic mechanism to layout connected components. (For more information about this mechanism, see *Layout of Connected Components* on page 25.) When using this mechanism, each component is laid out in its own individual level structure. Nodes of the first level of one component may be placed at a different position than nodes of the first level of another component.

The generic mechanism to layout connected components is, however, switched off by default. In this case, the layout algorithm can still handle disconnected graphs. It merges all components into a global level structure.

Link Clipping

The layout algorithm can use a link clip interface to clip the end points of a link. (See *Link Clipping* on page 27.)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip interface is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using a Link Clipping Interface* on page 151 for details of the link clipping mechanism.

Link Connection Box

The layout algorithm can use a link connection box interface (see *Link Connection Box* on page 29) in combination with the link clip interface. If no link clip interface is used, the link connection box interface has no effect. For details see *Using a Link Connection Box Interface* on page 150.

Percentage Completion Calculation

The layout algorithm calculates the estimated percentage of completion. This value can be obtained from the layout report during the run of the layout. (For a detailed description of this features, see *Percentage Completion Calculation* on page 31 and *Layout Event Listeners* on page 294.)

Preserve Fixed Links

The layout algorithm does not reshape the links that are specified as fixed. In fact, fixed links are completely ignored. (For more information on link parameters in the `IlvGraphLayout` class, see *Preserve Fixed Links* on page 32 and *Link Style* on page 109.)

Preserve Fixed Nodes

The layout algorithm does not move the nodes that are specified as fixed. (For more information on node parameters in the `IlvGraphLayout` class, see *Preserve Fixed Nodes* on page 32.) Moreover, the layout algorithm ignores fixed nodes completely and also does

not route the links that are incident to the fixed nodes. This can result in undesired overlapping nodes and link crossings. However, this feature is useful for individual, disconnected components that can be laid out independently.

Save Parameters to Named Properties

The layout algorithm is able to save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Save Parameters to Named Properties* on page 34 and *Saving Layout Parameters and Preferred Layouts* on page 315).

Stop Immediately

The layout algorithm stops after cleanup if the method `stopImmediately` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop Immediately* on page 35.) If the layout stops early because the allowed time has elapsed, the nodes and links are not moved from their positions before the layout call and the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific Parameters

The following parameters are specific to the `IlvHierarchicalLayout` class.

Flow Direction

The flow direction parameter specifies the direction in which the majority of the links should point. If the flow direction is to the top or to the bottom, the node levels are oriented horizontally and the links mostly vertically. If the flow direction is to the left or to the right, the node levels are oriented vertically and the links mostly horizontally.

If the flow direction is to the bottom, the nodes of the level with index 0 are placed at the top border of the drawing. The nodes with level index 0 are usually the root nodes of the drawing (that is, the nodes without incoming links). If the flow direction is to the top, the nodes with level index 0 are placed at the bottom border of the drawing. If the flow direction is to the right, the nodes are placed at the left border of the drawing.

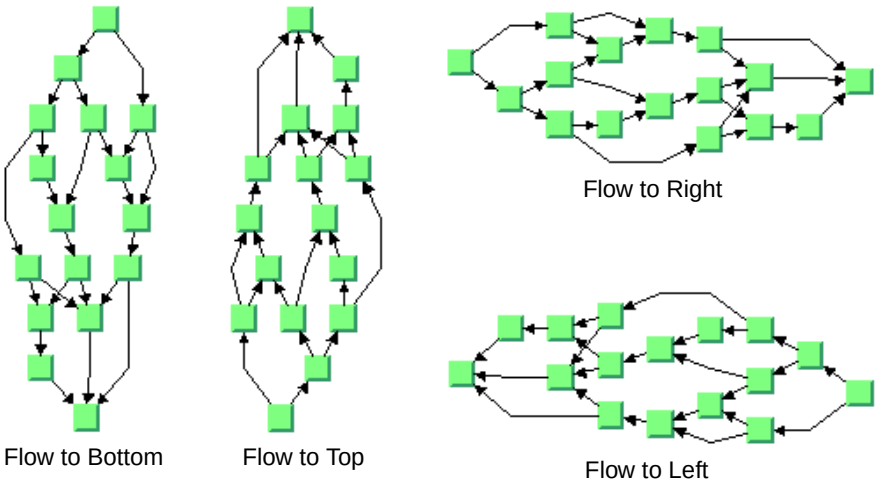


Figure 4.60 *Flow Directions*

To specify the flow direction, use the following method:

```
void setFlowDirection(int direction)
```

The valid values for `direction` are:

- `IlvDirection.Right` (the default)
- `IlvDirection.Left`
- `IlvDirection.Bottom`
- `IlvDirection.Top`

To obtain the current choice, use the following method:

```
int getFlowDirection()
```

Level Justification

If the layout uses horizontal levels, the nodes of the same level are placed approximately at the same y-coordinate. The nodes can be justified, depending on whether the top border, or the bottom border, or the center of all nodes of the same level should have the same y-coordinate.

If the layout uses vertical levels, the nodes of the same level are placed approximately at the same x-coordinate. In this case, the nodes can be justified to be aligned at the left border, at the right border, or at the center of the nodes that belong to the same level.

To specify the level justification, use the following method:

```
void setLevelJustification(int justification)
```

If the flow direction is to the top or to the bottom, the valid values for `justification` are:

- `IlvDirection.Top`
- `IlvDirection.Bottom`
- `IlvDirection.Center` (the default)

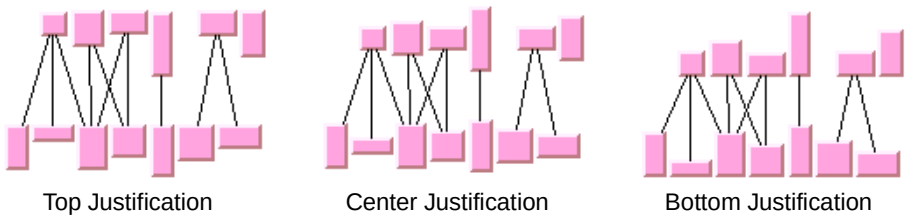


Figure 4.61 *Level Justification for Horizontal Levels*

If the flow direction is to the left or to the right, the valid values for `justification` are:

- `IlvDirection.Left`
- `IlvDirection.Right`
- `IlvDirection.Center` (the default)

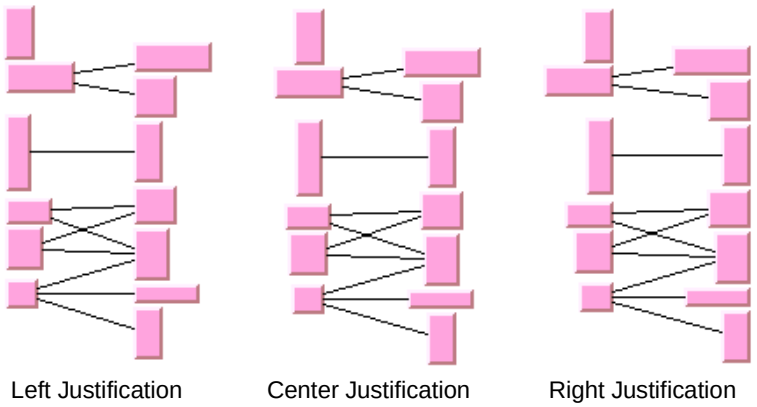


Figure 4.62 *Level Justification for Vertical Levels*

To obtain the current value, use the following method:

```
int getLevelJustification()
```

Link Style

The layout algorithm positions the nodes and routes the links. To avoid overlapping nodes and links, it creates bend points for the shapes of links. The link style parameter controls the

position and number of bend points. The link style can be set globally, in which case all links have the same kind of shape, or locally on each link such that different link shapes occur in the same drawing.

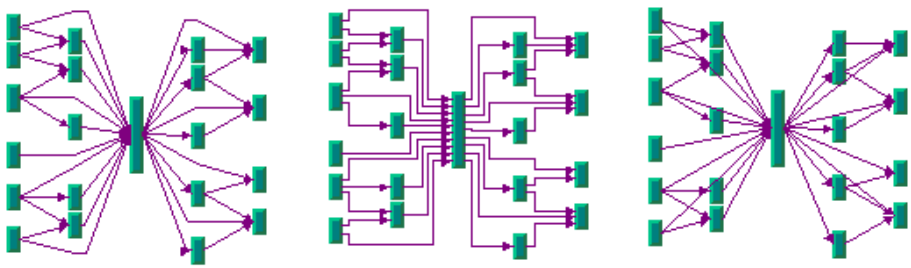


Figure 4.63 *Link Styles*

> ***Note:*** *The layout algorithm may raise an* `IlvInappropriateLinkException` *if the layout is performed on an* `IlvGrapher`*, but links are neither a subclass of* `IlvPolylineLinkImage` *nor of* `IlvSplineLinkImage`*, or if connectors are not a subclass of* `IlvRelativeLinkConnector`*. You can use the methods* `EnsureAppropriateLinkTypes`*,* `EnsureAppropriateLinkConnectors` *or* `EnsureAppropriateLinks` *defined in* `IlvGraphLayoutUtil` *to automatically replace inappropriate links or link connectors before layout or when the* `IlvInappropriateLinkException` *is caught. For details on these methods, see the ILOG JViews Graph Layout Reference Manual. For details on the graph model, see Using the Graph Model on page 297.*

Global Link Style

To set the global link style, use the following method:

```
void setGlobalLinkStyle(int style)
```

The valid values for `style` are:

- `IlvHierarchicalLayout.POLYLINE_STYLE`

  All links get a polyline shape. A polyline shape consists of a sequence of line segments that are connected at bend points. The line segments can be turned into any direction. This is the default value.

- `IlvHierarchicalLayout.ORTHOGONAL_STYLE`

  All links get an orthogonal shape. An orthogonal shape consists of orthogonal line segments that are connected at bend points. An orthogonal shape is a polyline shape where the segments can be turned only in directions of 0, 90, 180 or 270 degrees.

- `IlvHierarchicalLayout.STRAIGHT_LINE_STYLE`

  All links get a straight-line shape. All intermediate bend points (if any) are removed. This often causes overlapping nodes and links.

- `IlvHierarchicalLayout.NO_RESHAPE_STYLE`

  None of the links is reshaped in any manner. Note, however, that unlike fixed links, the links are not ignored completely. They are still used to calculate the leveling.

- `IlvHierarchicalLayout.MIXED_STYLE`

  Each link can have a different link style. The style of each individual link can be set such that different link shapes can occur in the same graph.

To obtain the current choice, use the following method:

```
int getGlobalLinkStyle()
```

Individual Link Style

All links have the same style of shape unless the global link style is `MIXED_STYLE`. Only when the global link style is `MIXED_STYLE` can each link have an individual link style.

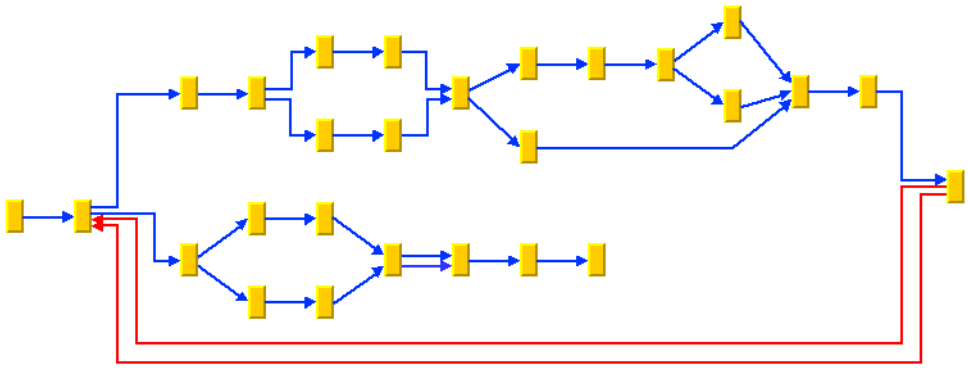


Figure 4.64 *Different Link Styles Mixed in the Same Drawing*

To set and retrieve the style of an individual line, use the following methods:

```
void setLinkStyle(Object link, int style)
```

```
int getLinkStyle(Object link)
```

The valid values for `style` for local links are the same as for the global link style:

- `IlvHierarchicalLayout.POLYLINE_STYLE`
- `IlvHierarchicalLayout.ORTHOGONAL_STYLE`
- `IlvHierarchicalLayout.STRAIGHT_LINE_STYLE`
- `IlvHierarchicalLayout.NO_RESHAPE_STYLE`

> ***Note:*** *The link style of a Hierarchical Layout graph requires links in an* `IlvGrapher` *that can be reshaped. Links of type* `IlvLinkImage`, `IlvOneLinkImage`, `IlvDoubleLinkImage`, `IlvOneSplineLinkImage`, *and* `IlvDoubleSplineLinkImage` *cannot be reshaped. You should use the class* `IlvPolylineLinkImage` *or* `IlvSplineLinkImage` *instead.*

Connector Style

The layout algorithm positions the end points of links (the connector pins) at the nodes automatically. The connector style parameter specifies how these end points are calculated.

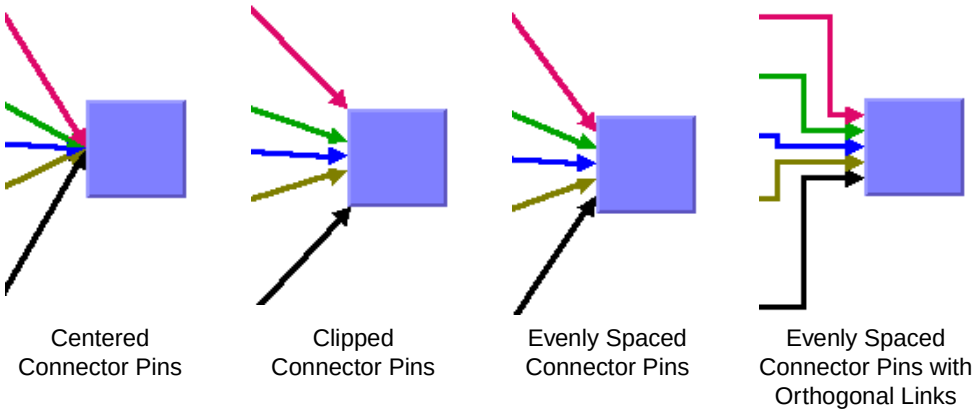


Figure 4.65 *Connector Styles*

To set the connector style, use the following method:

```
void setConnectorStyle(int style)
```

The valid values for `style` are:

- `IlvHierarchicalLayout.CENTERED_PINS`

  The end points of the links are placed in the center of the border where the links are attached. This option is well-suited for polyline links and straight-line links. It is less well-suited for orthogonal links, because orthogonal links can look ambiguous in this style.

- `IlvHierarchicalLayout.CLIPPED_PINS`

  Each link pointing to the center of the node is clipped at the node border. The connector pins are placed at the points on the border where the links are clipped. This option is particularly well-suited for polyline links without port specifications. It should not be used if a port side for any link is specified.

- `IlvHierarchicalLayout.EVENLY_SPACED_PINS`

  The connector pins are evenly distributed along the node border. This style guarantees that the end points of the links do not overlap. This is the best style for orthogonal links and works well for other link styles.

- `IlvHierarchicalLayout.AUTOMATIC_PINS`

  The connector style is selected automatically depending on the link style. If any of the links has an orthogonal style or if any of the links has a port side specification, the algorithm chooses evenly spaced connectors. If all the links are straight, it chooses centered connectors. Otherwise, it chooses clipped connectors.

To obtain the current choice, use the following method:

```
int getConnectorStyle()
```

> ***Note:*** *The connector style parameter requires link connectors at the nodes of an* `IlvGrapher` *that allow connector pins to be placed freely at the node border. It is recommended that you use* `IlvRelativeLinkConnector` *for link connectors to be used in combination with* `IlvGrapher` *objects.*

End Points Mode

Normally, the layout algorithm is free to choose the termination points of each link. However, if fixed-link connectors are used (for instance, `IlvPinLinkConnector`), the user can specify that the current fixed termination pin of a link should be used.

The layout algorithm provides two end point modes. You can set the end point mode globally, in which case all end points have the same mode, or locally on each link, in which case different end point modes occur in the same drawing.

Global End Point Mode

To set the global end point mode, the following methods are provided:

```
void setGlobalOriginPointMode(int mode);

void setGlobalDestinationPointMode(int mode);
```

The valid values for `mode` are:

- `IlvLinkLayout.FREE_MODE` (the default)

The layout is free to choose the appropriate position of the connection point on the origin/destination node.

- `IlvLinkLayout.FIXED_MODE`

  The layout must keep the current position of the connection point on the origin/destination node.

- `IlvLinkLayout.MIXED_MODE`

  Each link can have a different end point mode.

The connection points are automatically considered as fixed if they are connected to *grapher pins*.

To obtain the current choice, use the following methods:

```
int getGlobalOriginPointMode();

int getGlobalDestinationPointMode();
```

Individual End Point Mode

All links have the same end point mode unless the global end point mode is `IlvLinkLayout.MIXED_MODE`. Only when the global end point mode is set to `MIXED_MODE` can each link have an individual end point mode.

To set and retrieve the mode of an individual link, use the following methods:

```
void setOriginPointMode(Object link, int mode);

int getOriginPointMode(Object link);

void setDestinationPointMode(Object link, int mode);

int getDestinationPointMode(Object link);
```

The valid values for `mode` are:

- `IlvLinkLayout.FREE_MODE` (the default)
- `IlvLinkLayout.FIXED_MODE`

The connection points are automatically considered as fixed if they are connected to *grapher pins*.

> ***Note:*** *The layout algorithm may raise an* `IlvInappropriateLinkException` *if layout is performed on an* `IlvGrapher`*, but links are neither a subclass of* `IlvPolylineLinkImage` *nor of* `IlvSplineLinkImage`*, or if connectors are not a subclass of* `IlvRelativeLinkConnector`*. You can use the methods* `EnsureAppropriateLinkTypes`*,* `EnsureAppropriateLinkConnectors` *or* `EnsureAppropriateLinks` *defined in* `IlvGraphLayoutUtil` *to automatically replace inappropriate links or link connectors before layout or when the* `IlvInappropriateLinkException` *is caught. For details on these methods, see the ILOG JViews Graph Layout Reference Manual. For details on the graph model, see Using the Graph Model on page 297.*

Using a Link Connection Box Interface

By default, the connector style determines how the connection points of the links are distributed on the border of the bounding box of the nodes, symmetrically with respect to the middle of each side. Sometimes it may be necessary to place the connection points on a rectangle smaller or larger than the bounding box. For instance, this can happen when labels are displayed below or above nodes.

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkConnectionBoxInterface`. An example for the implementation of a link connection box interface is in *Link Connection Box* on page 29. To set a link connection box interface, call:

```
void setLinkConnectionBoxInterface(IlvLinkConnectionBoxInterface
interface)
```

The link connection box interface provides each node with a link connection box and tangential shift offsets. The Hierarchical Layout uses the link connection box but does not use the tangential offsets.

Figure 4.66 illustrates the effects of customizing the connection box. On the left is the result without any connection box interface. The picture on the right shows the effect if the connection box interface returns the dashed rectangle for the blue node.

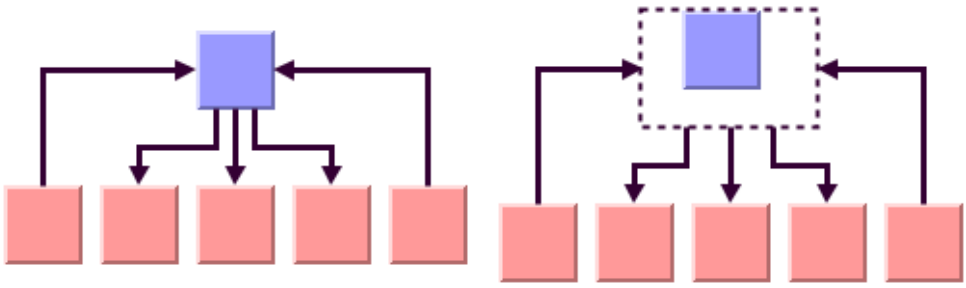


Figure 4.66 *Effect of Connection Box Interface*

Using a Link Clipping Interface

By default, the Hierarchical Layout places the connection points of links at the border of the bounding box of the nodes. If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape. An example is shown in Figure 4.67.

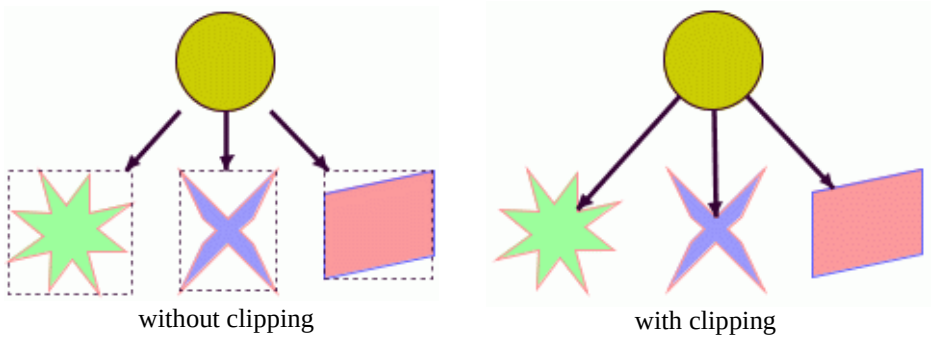


Figure 4.67 *Effect of Link Clipping Interface*

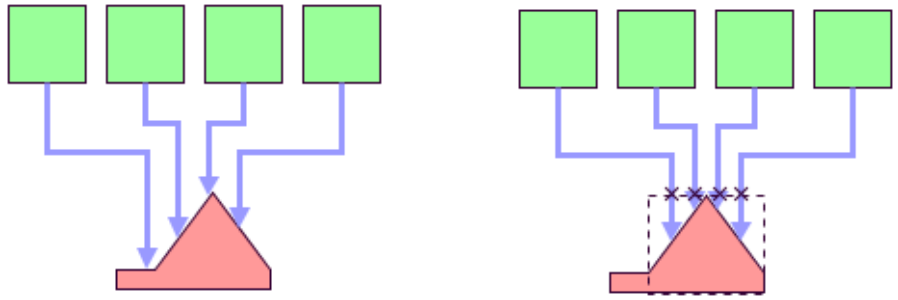
You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkClipInterface`. An example for the implementation of a link clip interface is in *Link Clipping* on page 27. To set a link clip interface, call:

```
void setLinkClipInterface(IlvLinkClipInterface interface)
```

> ***Note:*** *Additionally to the link clip interface, the* `IlvClipLinkConnector` *can be used. This special link connector updates the clipped connection points automatically during interactive node movements.*

The connector style, the link connection box interface, and the link clip interface work together in the following way: by respecting the connector style, the proposed connection points are calculated on the rectangle obtained from the link connection box interface (or on the bounding box of the node, if no link connection box interface was specified). Then, the proposed connection point is passed to the link clip interface and the returned connection points are used to connect the link to the node.

An example of the combined effect is shown in Figure 4.68.



Clipping at the node bounding box

Clipping at a specified connection box

Figure 4.68 *Combined Effect of Link Clipping Interface and Link Connection Box*

If the links are clipped at the red node in Figure 4.68 (left), they appear unsymmetric with respect to the node shape, because the relevant part of the node (here: the triangle) is not in the center of the bounding box of the node, but the proposed connection points are calculated with respect to the bounding box. This can be corrected by using a link connection box interface to explicitly specify a smaller connection box for the relevant part of the node (Figure 4.68 right) such that the proposed connection points are placed symmetrically at the triangle of the node.

For Experts: Thick Links

If evenly spaced pins are used as connector style, the links can be evenly spaced with respect to the link center or with respect to the link border. The difference is only visible when links that connect to the same node have different widths. For instance, when the link width indicates the cost or capacity of a flow in the application, many different link width may occur.

Figure 4.69 shows the effect of using different link widths. In the drawing on the left, the center of the links are evenly distributed at the left node. Each link has the same space

available at the node side. Therefore, the thick links appear closer to each other than do the thinner links and the offsets between the link borders are different. In the drawing on the right, the thick links have more space available than do the thinner links. The offset between the link border (at the segments that connect to the left node) is constant because the link width is considered in the calculation of the connection points.

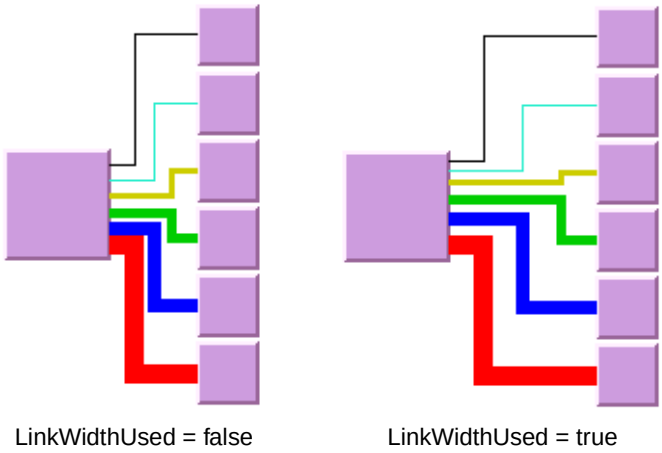


Figure 4.69 *Using the Link Width*

To enable the connector calculation to respect the link width, use the following call:

```
layout.setLinkWidthUsed(true);
```

The link width setting is disabled by default. The link width has no effect if the connector styles `CENTERED_PINS` or `CLIPPED_PINS` are used.

Port Sides Parameter

The Hierarchical Layout algorithm produces a layout where the majority of the links flow in the same direction. If the flow direction is towards the bottom, usually the incoming links are connected to the top side of the node and the outgoing links are connected to the bottom side of the node. It is also possible to specify on which side a link connects to the node.

To simplify the explanations of the port sides, we use the compass directions *north, south, east,* and *west*. The specified link flow direction is always towards south and the first level is towards north. If the flow direction is towards bottom, north is at the top, south at the bottom, east on the right, and west on the left side of the drawing. If the flow direction is towards right, north is on the left, south on the right, east at the top, and west at the bottom.

Figure 4.70 shows a drawing where the links connect to the larger middle node at the specified port sides. A compass icon shows the compass directions in these drawings.

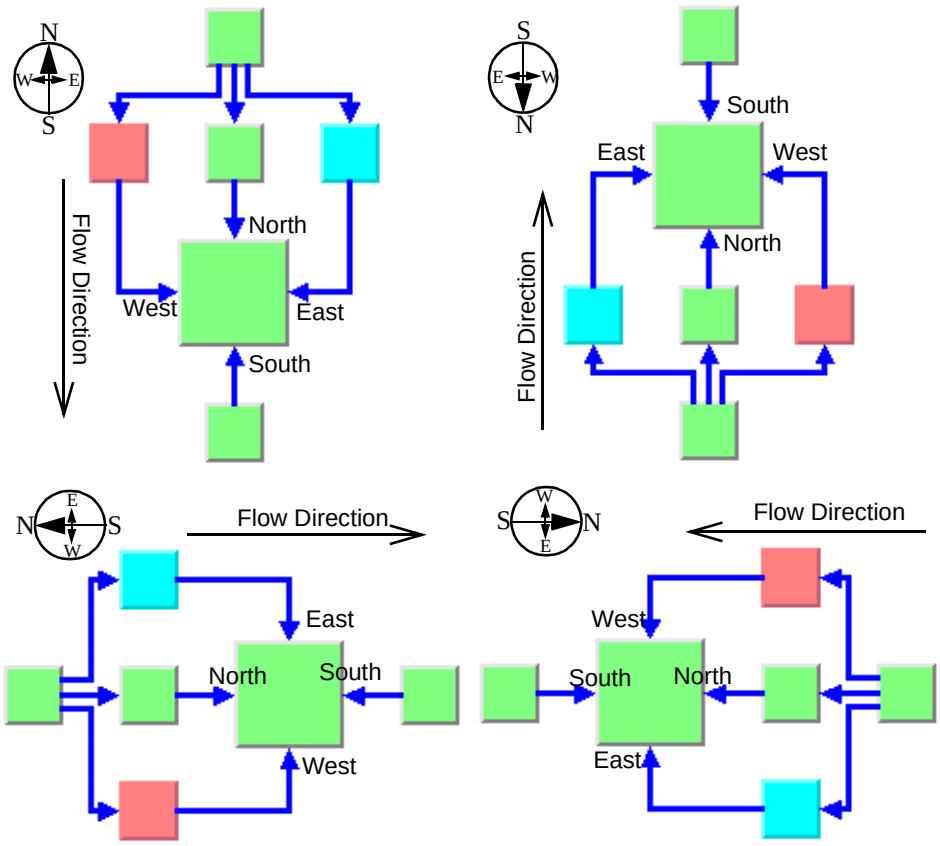


Figure 4.70 *Link Connections to Port Sides*

To specify for a link at which side the link connects to its source node, use the method:

```
void setFromPortSide(Object link, int side);
```

To specify for a link at which side the link connects to its destination node, use the method:

```
void setToPortSide(Object link, int side);
```

The valid values for `side` are:

- `IlvHierarchicalLayout.UNSPECIFIED` (the default)
- `IlvHierarchicalLayout.NORTH`
- `IlvHierarchicalLayout.SOUTH`
- `IlvHierarchicalLayout.EAST`
- `IlvHierarchicalLayout.WEST`

To retrieve the current choice for a link, use the following methods:

```
int getFromPortSide(Object link);

int getToPortSide(Object link);
```

The port sides east and west work particularly well with the orthogonal link style. Polyline links with these port sides sometimes have unnecessary bends. Furthermore, if port sides are specified, the connector style `CLIPPED_PINS` should not be used.

Port Index Parameter

Instead of asking the layout algorithm to decide at which point a link connects to the node border, you can specify where the links connect to the node. You cannot specify the exact location, but you can specify the relative location compared to the connection points of the other links. This is done by using a port index. Figure 4.58 on page 136 shows a sample layout with ports at many nodes.

Links that have the same port index connect at the same point of the node. The ports are evenly distributed at the node sides, in a similar way as with the connector style `EVENLY_SPACED_PINS`.The ports are ordered according to their indices. On the north and south side of a node, the port indices increase toward the east. On the east and west sides of a node, the port indices increase toward the south. By using port indices in this way, it is easier to rotate a graph by simply changing the flow direction without needing to update all the port specifications.

Figure 4.71 show how the port indices depend on the flow direction.

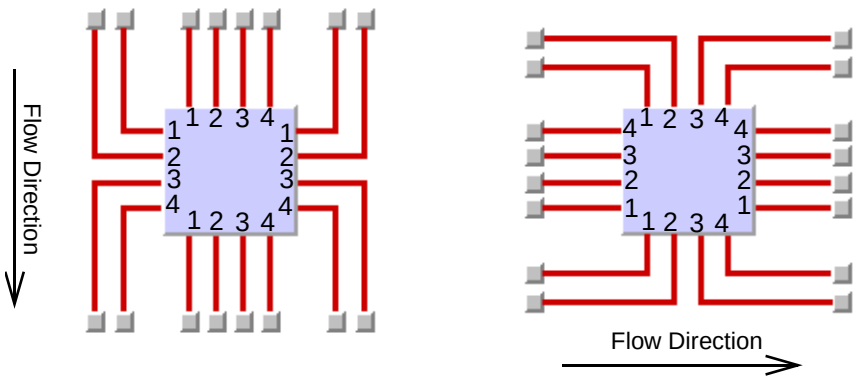


Figure 4.71 *Port Index Numbering Conventions in Relation to Flow Direction*

Port numbers are normally used in combination with port sides. Therefore, you must specify how many ports are available on each side of a node. To specify the number of ports, use the following method:

```
void setNumberOfPorts(Object node, int side, int numberOfPorts);
```

For example, to use 4 ports on each side of a specific node, use the following calls:

```
layout.setNumberOfPorts(node, IlvHierarchicalLayout.EAST, 4);
layout.setNumberOfPorts(node, IlvHierarchicalLayout.WEST, 4);
layout.setNumberOfPorts(node, IlvHierarchicalLayout.NORTH, 4);
layout.setNumberOfPorts(node, IlvHierarchicalLayout.SOUTH, 4);
```

The node side is specified again by `EAST`, `WEST`, `NORTH`, and `SOUTH`. To retrieve the retrieve the number of ports available at the node, use the following method:

```
int getNumberOfPorts(Object node, int side);
```

After the number of ports per side is specified, you can choose which port each link connects to. You choose the port side and the port index for a link as follows:

To specify the connection at the source node, use the following methods:

```
void setFromPortSide(Object link, int portSide);

void setFromPortIndex(Object link, int portIndex);
```

To specify the connection at the destination node, use the following methods:

```
void setToPortSide(Object link, int portSide);

void setToPortIndex(Object link, int portIndex);
```

To obtain the current port index of a link, use the following methods:

```
int getFromPortIndex(Object link);

int getToPortIndex(Object link);
```

Using the port side and port index specifications are additional constraints for the layout algorithm. The more constraints are specified, the more difficult it is to calculate a layout. Therefore, if too many links have a specified port index, this resulting layout may have more link crossings and be less balanced.

Fork Link Shapes

If several links start at the same position and are orthogonally routed, it is sometimes preferred that the links share the first two link segments. The shape of a link bundle of this kind looks like a fork. In order to enable the fork shape mode for outgoing links, call:

```
layout.setFromFork(true);
```

In order to enable the fork shape mode for incoming lins, call:

```
layout.setToFork(true);
```

These statements have an effect only if the links are routed orthogonally. The fork appears only at those links that start or end exactly at the same point. Specifying `setFromFork(true)` by itself does not force the links to start at the same point. In order to force links to start or end at the same point, use the center connector style (see *Connector*

Style on page 147) or specify the same port for the links (see *Port Index Parameter* on page 155).

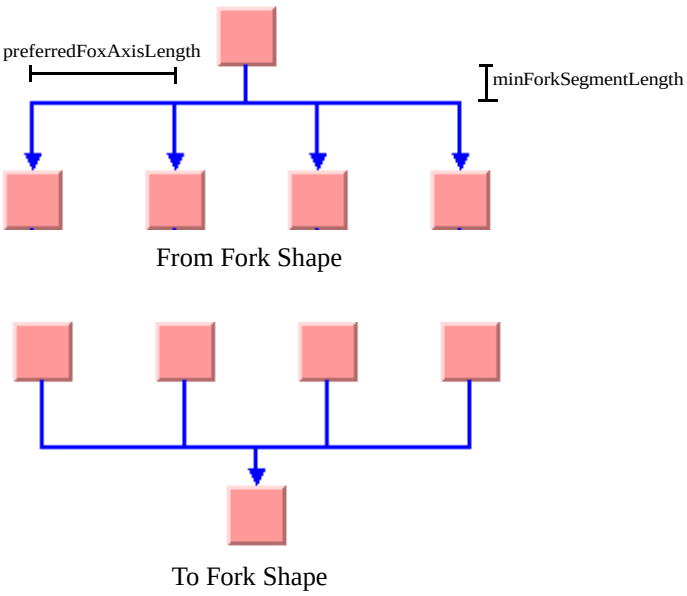


Figure 4.72 *Fork Link Shapes*

There are two spacing parameters for the fork shape:

```
void setMinForkSegmentLength(float length)
```

It sets the minimal length of the segment that is directly adjacent to the node.

```
void setPreferredForkAxisLength(float length)
```

This method sets the preferred length of the fork axis per branch (the second segment adjacent to the node). If the fork has five branches, the entire axis has the preferred length five times the specified parameter. The preferred fork axis length is only a hint for the layout algorithm. If enough space is available, the algorithm will enlarge the fork axis to avoid unnecessary link bends. If there is not enough space, the algorithm may as well calculate a fork axis that is smaller than the preferred one.

Fork link shapes may sometimes look ambiguous, in particular when a link starts at the same point where another link ends, because in this case it is impossible to recognize whether the arrowhead belongs to one or the other link.

Link Priority Parameter

The layout algorithm tries to place the nodes such that all links are short, point in the flow direction, and do not cross each other. However, this is not always possible. Often, links

cannot have the same length. If the graph has cycles, some links must be reversed against the flow direction. If the graph is nonplanar, some links have to cross each other.

The link priority parameter controls which links should be selected if long, reversed, or crossing links are necessary. Links with a low priority are more likely to be selected than links with a high priority. This does not mean that low-priority links are always longer, reversed, or crossed, because the graph may have a structure such that no long, reversed or crossing links are necessary.

To set or retrieve the link priority, use the following methods.

```
void setLinkPriority(Object link, float priority)

float getLinkPriority(Object link)
```

The default value of the link priority is 1.0. Negative link priorities are not allowed.

For an example of using the link priority, consider a cycle A->B->C->D->E->A. It is impossible to lay out this graph without reversing any link. Therefore, the layout algorithm selects one link to be reversed. To control which link is selected, you can give one link a lower priority than the others. This link will be reversed. In Figure 4.73, the bottom layout shows the use of the link priority. The link C->D was given the priority 0.5, while all the other links have the priority 1.0. Therefore C-D is reversed. The top layout in Figure 4.73 shows what happens when all links have the same priority. Link E->A is reversed.

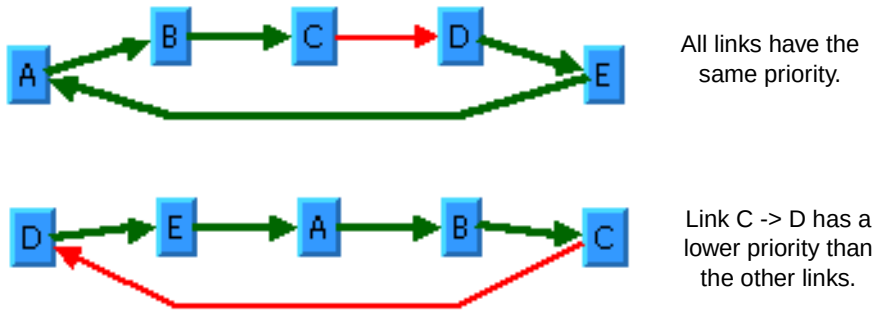


Figure 4.73 *Working with Link Priorities*

The use of link priorities is important in combination with ports. Links with “from” ports on the south side and “to” ports on the north side are preferably laid out opposite to the flow direction. Such a feedback link may cause parts of the drawing to tip over. Figure 4.74 shows an example. The red link is a feedback link with port specifications. To obtain the correct result as shown in the drawing on the right side of Figure 4.74, you would set the priority of the feedback link to a very low value.

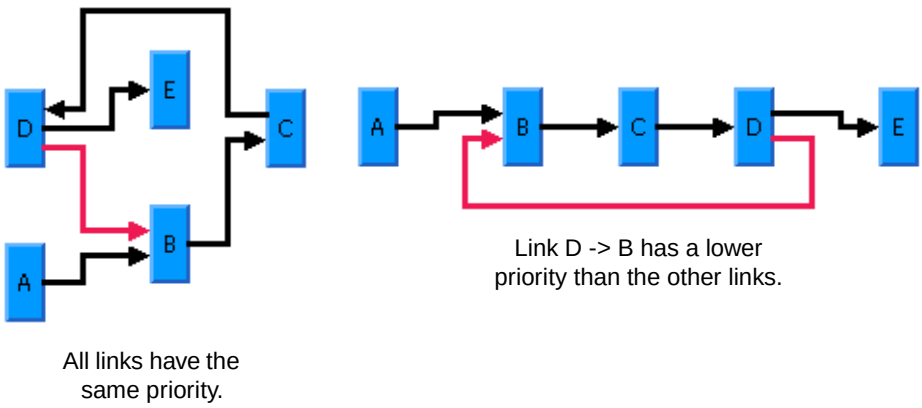


Figure 4.74 *Using Link Priorities and Ports*

Spacing Parameters

The spacing of the layout is controlled by three kinds of spacing parameters: the minimal offset between nodes, the minimal offset between parallel segments of links and the minimal offset between a node border and a bend point of a link or a link segment that is parallel to this border. The offset between parallel segments of links is at the same time the offset between bend points of links. All three kind of parameters occur in both directions: horizontally and vertically.

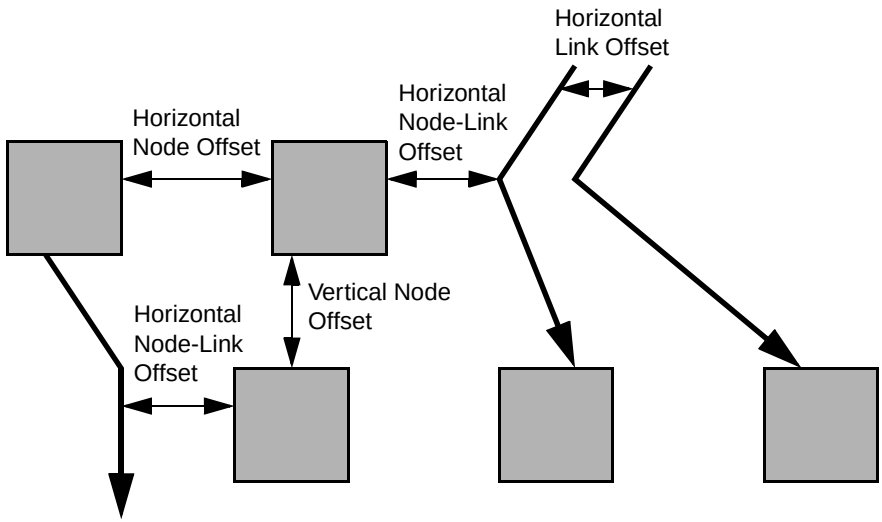


Figure 4.75 *Spacing Parameters*

The spacing parameters can be set for the horizontal direction by the following methods:

```
void setHorizontalNodeOffset(float offset)
void setHorizontalLinkOffset(float offset)
void setHorizontalNodeLinkOffset(float offset)
```

The spacing parameters can be set for the vertical direction by the following methods:

```
void setVerticalNodeOffset(float offset)
void setVerticalLinkOffset(float offset)
void setVerticalNodeLinkOffset(float offset)
```

The spacing parameters can be obtained by the corresponding methods:

```
float getHorizontalNodeOffset()
float getHorizontalLinkOffset()
float getHorizontalNodeLinkOffset()
float getVerticalNodeOffset()
float getVerticalLinkOffset()
float getVerticalNodeLinkOffset()
```

For a layout with horizontal levels (the flow direction is to the top or to the bottom), the horizontal node offset is the minimal distance between nodes of the same level. The vertical node offset is the minimal distance between nodes of different levels, that is, the minimal distance between the levels. For non-orthogonal link styles, the horizontal link offset is basically the minimal distance between bend points of links. The horizontal node-link offset is the minimal distance between the node border and the bend point of a link. For horizontal levels, the vertical link offset and the vertical node-link offset play a role only if the link shapes are orthogonal.

Similarly, for a layout with vertical levels (the flow direction is to the left or to the right), the vertical node offset controls node distances within the levels. The horizontal node offset is the minimal distance between the levels. In this case, the vertical link offset and the vertical node-link offset always play a role, while the horizontal link offset and the horizontal node-link offset affect the layout only with orthogonal links.

For orthogonal links, the horizontal link offset is the minimal distance between parallel, vertical link segments. The vertical link offset is the minimal distance between parallel, horizontal link segments. However, the layout algorithm cannot always satisfy these offset requirements. If a node is very small but has many incident links, it may be impossible to place the links orthogonally with the specified minimal link distance on the node border. In this case, the algorithm places some link segments closer than the specified link offset.

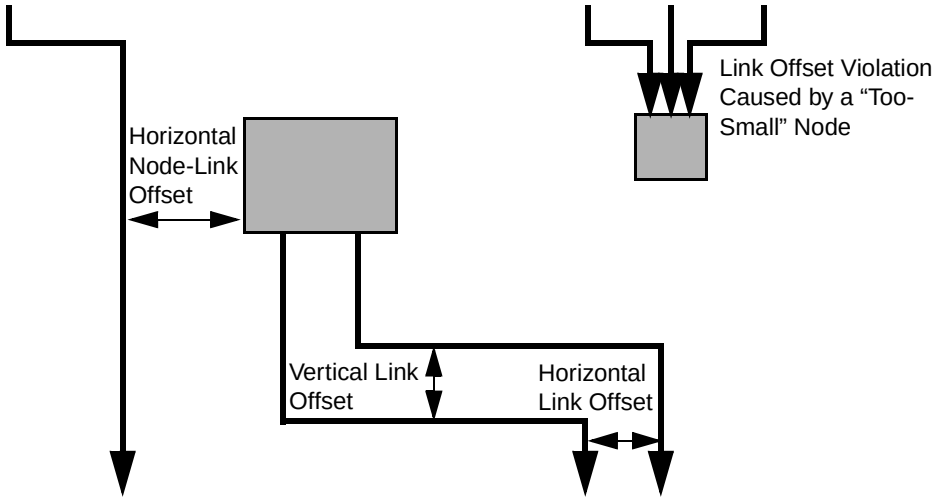


Figure 4.76 *Spacing Parameters for Orthogonal Links*

Incremental Mode

In some circumstances you may need to use a sequence of layouts on the same graph. For example:

- You work with graphs that have become out-of-date and you need to extend the graph. If you perform a layout on the extended graph, you probably want to identify the parts that were already laid out in the original graph. The layout should not change very much when compared with the layout of the original graph.
- The first layout results in a drawing with minor deficiencies. You want to solve these deficiencies manually and perform a second layout to clean up the drawing. The second layout probably should not greatly change the parts of the graph that were already acceptable after the first layout.

The Hierarchical Layout normally works nonincrementally. It performs a layout from scratch and moves all nodes to new positions and reroutes all links. The previous positions of nodes have no influence on the result of the layout. Hence, even a small change can cause a large effect on the next layout.

But the Hierarchical Layout also supports incremental sequences of layout that “do not change very much.” It can place the nodes close to their previous positions, so that you can more easily identify the parts that had already been laid out in the original graph. Incremental mode takes the previous positions of the nodes into account. In this mode the algorithm preserves the relative order of the levels and the nodes within the levels in the subsequent layout. It does not preserve the absolute positions of the nodes, but it tries to

detect the structure of the previous layout by examining the node coordinates. For instance, if two nodes are in the same level, then they stay in the same level after an incremental layout. If a node is in a higher level than another node, it stays in the higher level.

Figure 4.77 on page 163 illustrates the difference between an incremental and nonincremental layout.

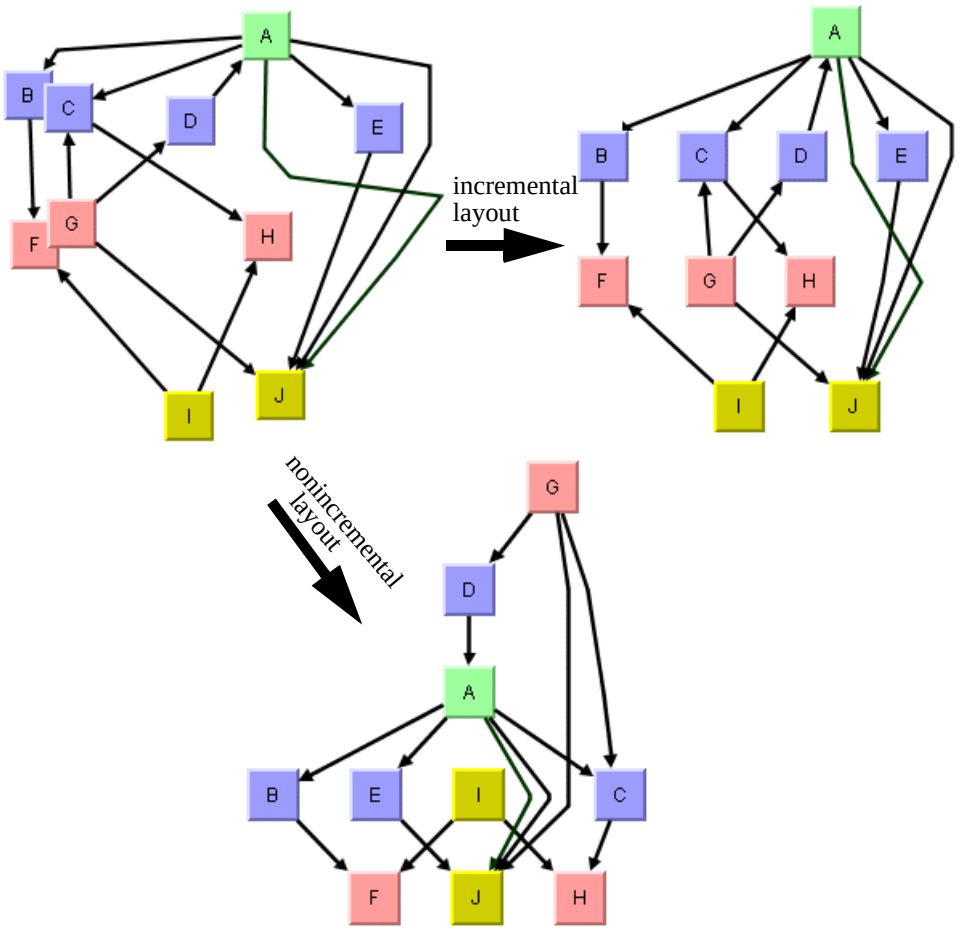


Figure 4.77 *Incremental and Nonincremental Layouts*

Incremental mode is disabled by default. To enable incremental mode, call:

```
layout.setIncrementalMode(true);
```

You can determine whether incremental mode is enabled by calling:

```
layout.isIncrementalMode();
```

Phases of the Incremental Mode

The layout algorithm analyzes the drawing in incremental mode in the following way:

1. First, it determines from the node coordinates which nodes must belong to the same level. For instance, if the flow direction is towards the bottom, it tries to detect horizontal reference lines at those vertical positions where many nodes are placed along a line. The specified vertical node offset helps to detect these lines because the horizontal reference lines should be approximately the vertical node offset apart. See Figure 4.78 on page 164.
2. All nodes that touch the same reference line are assigned to the same level.
3. It determines the order of the nodes within each level by analyzing where the node touches the reference line. For instance, if the flow direction is towards the bottom, it determines from the x coordinate of the nodes how they are ordered within the levels.
4. If long links span several levels, the algorithm can preserve the shape of a long link. It determines the point where a link crosses the level reference line. It creates a bend point for the long link inside the level. It tries to preserve the order of the bend points in each level. For instance, if in a flow direction towards the bottom, a long link bypasses another node on the right side, then the incremental layout tries to find a similar shape of the link that bypasses the node on the right side, as illustrated in Figure 4.78.
5. Finally, the layout tries to calculate the absolute positions of the nodes that respect the levels and the ordering within the levels. It tries to balance the node positions. However, it also tries to place each node close to its previous position. Both criteria often compete with each other, because in order to get a perfect balance, nodes must sometimes move far from their original position. The Hierarchical Layout contains a parametrized heuristic to satisfy both criteria.

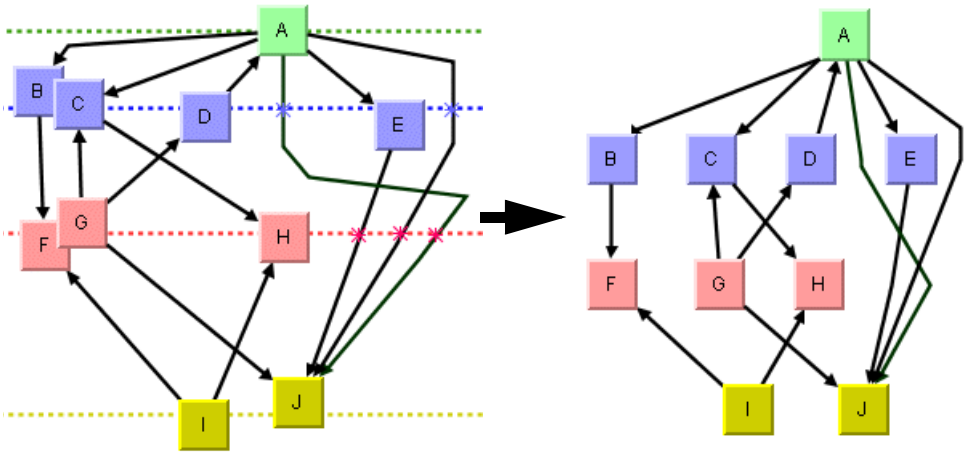


Figure 4.78 *Incremental Layout Phases*

Expert Parameters of the Incremental Mode

Each phase of the incremental mode can be parameterized. These layout parameters have an effect only if incremental mode is switched on.

Minimizing Long Link Crossings

As discussed in *Phases of the Incremental Mode* on page 163, the incremental layout tries to preserve the shape of long links that cross several levels. This implies that link crossings between long links are not resolved. If crossings of long links are not desired, it may be better to reroute long links from scratch. Figure 4.79 has four hierarchy trees, with the original layout at the upper left. The bottom right shows the result if long links are rerouted, and the top right shows the result if the shape of long links are preserved.

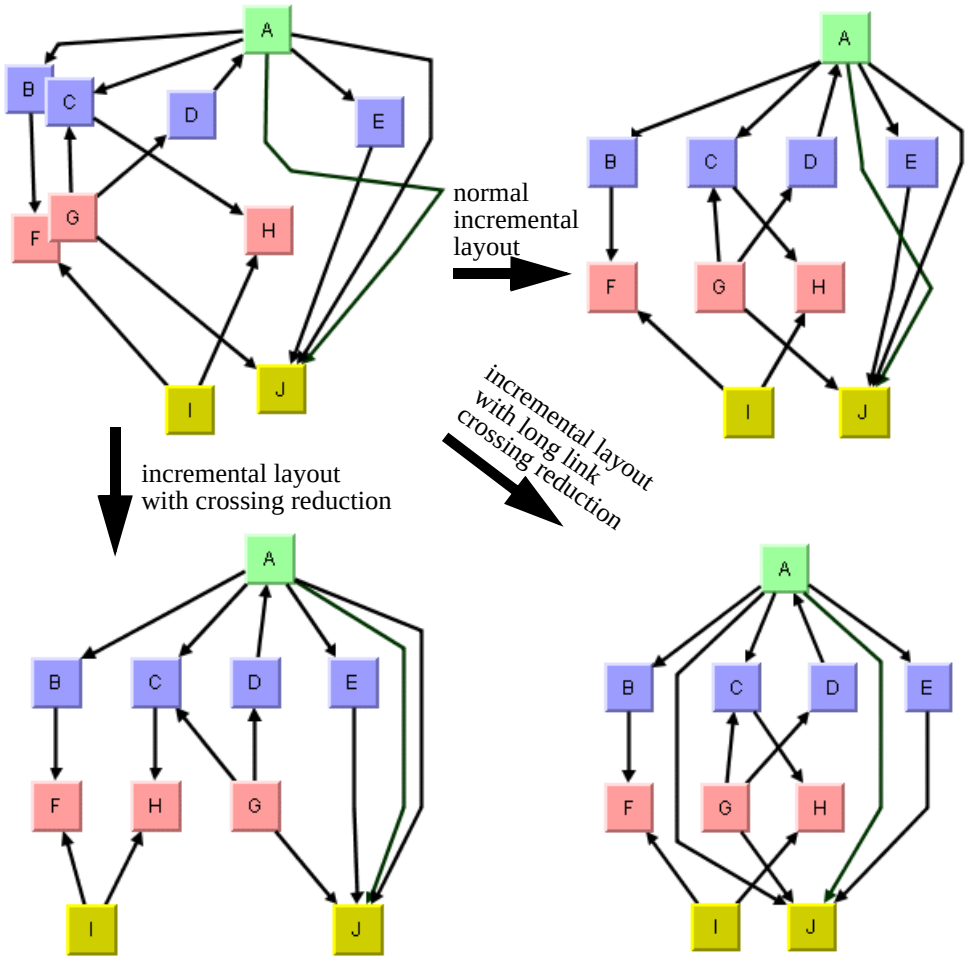


Figure 4.79 *Crossing Reduction During Incremental Layouts*

To reroute long links from scratch, you must enable the crossing reduction mechanism for long links, by the statement:

```
layout.setLongLinkCrossingReductionDuringIncremental(true);
```

The crossing reduction of long links determines only the shape of the links. It does not influence the order of the other nodes within the levels.

Minimizing All Link Crossings

Optionally, you can apply a crossing reduction to all nodes within each level. In this case, the incremental layout determines from the node coordinates which nodes belong to the same level, but it may reorder the nodes within the levels completely to avoid link crossings. It

also reorders the long links in this case. Figure 4.79 on page 166, bottom left shows the result. Notice that the order of the nodes “F,” “G,” and “H” have changed to resolve the link crossings.

To enable the crossing reduction for all nodes, call:

```
layout.setCrossingReductionDuringIncremental(true);
```

Setting Absolute Level Positioning

As discussed in *Phases of the Incremental Mode* on page 163, the incremental layout tries to place the nodes in absolute positions that are close to the previous positions. It tries to avoid nodes moving a large distance, because even if the relative order of the nodes within the levels does not change, large movement distances can be confusing for users. It is much easier to keep a mental map of the diagram if the nodes remain close to the previous positions.

Figure 4.80 on page 168 illustrates node repositioning with and without taking the previous positions into account. The incremental layout of the original graph at the top left results in the graph at the top right, which is easier to recognize as the same graph than the graph at the bottom.

The absolute level positioning feature is enabled by default, but it can be disabled by calling:

```
layout.setIncrementalAbsoluteLevelPositioning(false);
```

With this statement, the layout does not try to place the nodes close to the previous positions. It places the nodes such that the layout is balanced. However, to create a perfect balance, the layout may need to move a few nodes so far apart that you can no longer recognize the diagram after the layout from the node positions that were shown in the previous layout (see Figure 4.80, bottom).

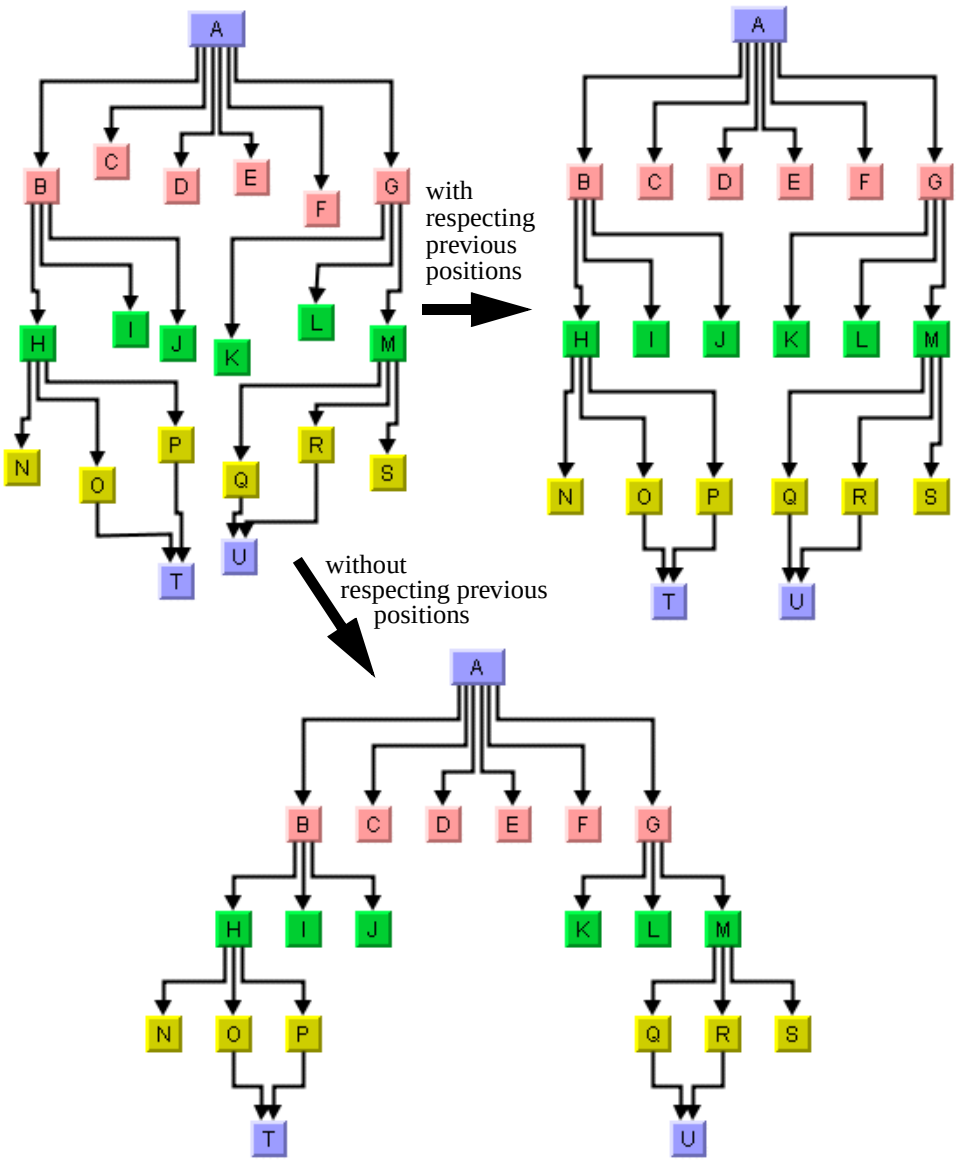


Figure 4.80 *Absolute Positioning During Incremental Layouts*

Setting Absolute Level Position Range and Tendency

If absolute level positioning is enabled, it competes with the aesthetic criteria to create a balanced layout. Due to the fact that nodes must stay close to their previous positions, the diagram after incremental layout may be somewhat unbalanced and unsymmetrical. The

Hierarchical Layout algorithm uses a heuristic that you can influence by two parameters, the absolute level position range and tendency.

Set the absolute level position range as in the following example:

```
layout.setIncrementalAbsoulteLevelPositionRange(100);
```

This statement specifies that within the range of 100 coordinate units from the old position of the node, the balance is the only criteria for the placement. This means that a node whose optimal position is less than 100 coordinate units away from its previous position is placed exactly at its optimal position. Nodes whose optimal position is farther away are placed at a position that is a compromise between previous position and optimal position. This is illustrated in Figure 4.81 on page 170 right.

Set the absolute level position tendency as in the following example:

```
layout.setIncrementalAbsoluteLevelPositionTendency(70);
```

This statement specifies that positions of nodes whose optimal positions are far away from their previous position are 70% influenced by their previous position and 30% influenced by their optimal positions. Imagine a rubber band that tries to pull a node to its previous position, and another rubber band that tries to pull the same node to its optimally balanced position. The level position tendency `70` means that one rubber band pulls with 70% of the force towards the previous position, and the other rubber band pulls with 30% towards the optimal position. Increasing the tendency means that the node stays closer to its old position, decreasing it means that the node moves closer to its optimal position. If you set the tendency to 0%, this has the same effect as disabling the incremental absolute level positioning (see Figure 4.81).

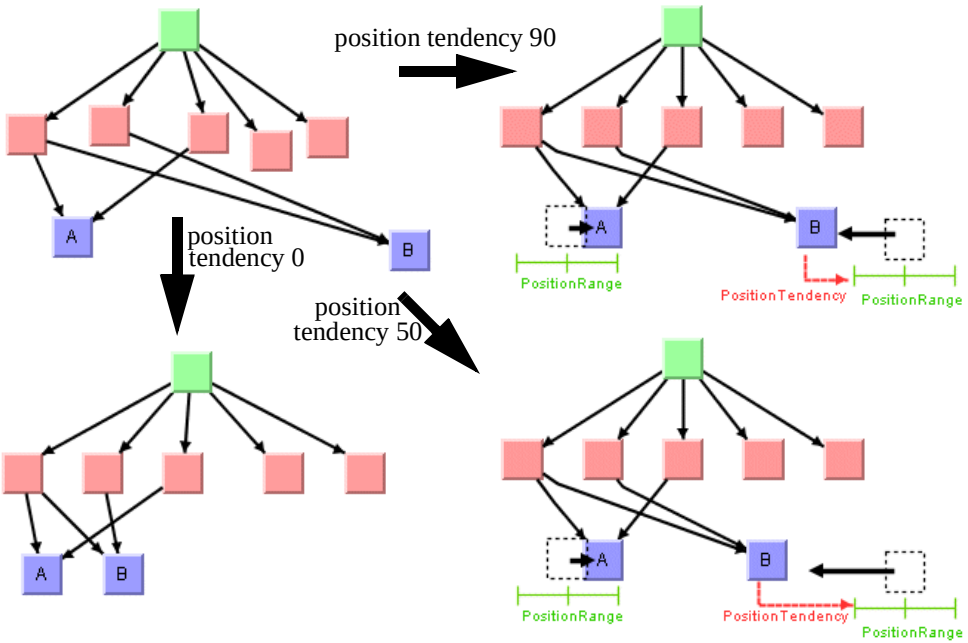


Figure 4.81 *Absolute Positioning During Incremental Layouts*

Marking Nodes for Incremental Layout

Incremental layout normally treats all nodes and links of the drawing in the same way. However, you may have added nodes and links to the drawing programmatically, and the new nodes and links do not have meaningful coordinates yet. Perhaps you have placed them all at the origin (0,0), or at random coordinates. In this case, you need an incremental layout that takes the coordinates of all nodes into account that were previously laid out, while it ignores the coordinates of all new nodes. The incremental mode of the Hierarchical Layout allows you to specify which nodes cannot be laid out incrementally, by calling:

```
layout.markForIncremental(nodeOrLink);
```

If you call this statement, the node or link is marked such that its coordinates are ignored during the next incremental layout. The positions of marked nodes and links are calculated from scratch. The mark is valid only until the next layout and is automatically cleared afterwards.

Layout Constraints

The Hierarchical Layout algorithm supports *relative position constraints* on nodes. Such a constraint is a rule on how a particular node (or a group of nodes) must be placed with respect to the other nodes. The constraints influence the relative positions. For example, you

can force node A to be on the left side of node B, so the position of A is expressed relative to the position of B. It is theoretically possible to specify contradicting constraints: if you specify that node A must be on the left side of B and B must be on the left side of A , then these constraints are not solvable at the same time. If A is on the left side of B, then B must be on the right side of A . The Hierarchical Layout algorithm tries to detect and resolve constraint conflicts automatically. It ignores those constraints that are infeasible. Since the automatic constraint resolution is time consuming, it is recommended to specify nonconflicting constraints when possible.

Constraints should be used only if the incremental mode is switched off. In fact, the incremental mode is implemented by means of additional constraints that are added internally. Hence, if you use constraints during the incremental mode, it is very likely that the system detects so many constraint conflicts that you get unexpected results.

Constraints should be used carefully. The more constraints are specified, the more difficult it is to calculate a layout. Therefore, this resulting layout may have more link crossings and be less balanced than a graph with no constraints.

Each type of constraint is represented by a subclass of `IlvHierarchicalConstraint`. The following constraint types are available:

`IlvLevelRangeConstraint`	Forces a node into a range of certain levels
`IlvSameLevelConstraint`	Forces two nodes to the same level.
`IlvRelativeLevelConstraint`	Forces a node to a lower/higher level than another node.
`IlvGroupSpreadConstraint`	Forces a group of nodes on levels that are no more than a specified spread value apart.
`IlvRelativePositionConstraint`	Forces a node to a lower/higher position than another node of the same level.
`IlvSideBySideConstraint`	Forces two nodes of the same level to be placed side by side.
`IlvExtremityConstraint`	Forces a node to the first or last level, or to the first or last position within a level.
`IlvSwimLaneConstraint`	Forces a group of nodes into the same rectangular swim lane area.

The uses of these classes are explained in the following topics.

- *Adding and Removing Constraints*
- *Node Groups*
- *Level Range Constraints*
- *Level Index Parameter*

- *Same Level Constraints*
- *Group Spread Constraint*
- *Relative Level Constraint*
- *Position Index Parameter*
- *Relative Position Constraints*
- *Side-By-Side Constraints*
- *Extremity Constraints*
- *Constraint Priorities*
- *For Experts: Constraint Validation*

Adding and Removing Constraints

To add a constraint to hierarchical layout, you allocate a new constraint object and call on the `IlvHierarchicalLayout` instance:

```
void addConstraint(IlvHierarchicalConstraint constraint);
```

You can add as many constraints as you want. The constraints will be respected during the following calls of layout, until you remove the constraints.To remove the constraint that was most recently added, call:

```
void removeConstraint();
```

To remove a specific constraint, call:

```
void removeConstraint(IlvHierarchicalConstraint constraint);
```

To remove all constraints that were previously added, call:

```
void removeAllConstraints();
```

You can retrieve the constraints that were added to Hierarchical Layout:

```
Enumeration getConstraints()
```

Node Groups

Some constraints affect single nodes. Other constraints affect groups of nodes. The class `IlvNodeGroup` is a convenient way to specify a group of nodes. You can create a group of nodes in the following way:

```
group = new IlvNodeGroup();
while (...) {
    group.add(node);
}
```

A node group has a similar functionality as a vector. You can ask for the size and elements of the group, remove elements from the group, or check whether a node already belongs to the group. You can also convert a vector of nodes into a group:

`group.add(node)`	Adds a node to the group.
`group.remove(node)`	Removes a node from the group.
`group.contains(node)`	Checks whether a node is in the group.
`group.size()`	Returns the number of nodes in the group.
`group.elements()`	Returns the nodes of the group as an Enumeration.
`group = new IlvNodeGroup(vector)`	Creates a new group that contains the nodes stored in the input vector.

Level Range Constraints

In Step 1 of the layout algorithm (the leveling phase), the nodes are partitioned into levels. These levels are indexed starting from 0. For instance, when the flow direction is to the bottom, the nodes of the level index 0 are placed at the topmost horizontal level line and the nodes with larger level index are placed at a position lower than the nodes with smaller level index (see Figure 4.59 on page 139). The layout algorithm calculates these level indices automatically.

You can affect how the levels are partitioned by specifying the range of the level index for some nodes. The nodes are placed in those levels with index in the specified range. You have to specify the minimal and maximal index of the desired level. For instance:

```
layout.addConstraint(new IlvLevelRangeConstraint(node, 5, 7));
```

forces the node to be placed between minimal level 5 and maximal level 7, that is, either in level 5, level 6, or level 7. If you want to place the node exactly at level 5, call:

```
layout.addConstraint(new IlvLevelRangeConstraint(node, 5, 5));
```

Alternatively, you can call:

```
layout.setSpecLevelIndex(node, 5);
```

which has exactly the same meaning.

If you want to force the node to level 5 and above, set `UNSPECIFIED` as the maximal level:

```
layout.addConstraint(
  new IlvLevelRangeConstraint(node, 5, IlvHierarchicalLayout.UNSPECIFIED));
```

If you want to force the node to level 5 and below (that is, level 0, ..., 5), set `UNSPECIFIED` as the minimal level:

```
layout.addConstraint(
  new IlvLevelRangeConstraint(node, IlvHierarchicalLayout.UNSPECIFIED, 5));
```

In this particular case, you could also use zero (0) as the minimal level because the level indices start at 0.

You can apply the constraint to a group of several nodes at once. This has the same effect as specifying the constraint for each single node of the group, but it is more memory efficient and convenient. For instance, if you want to force the group of nodes to the levels between 5 and 7, you can pass an `IlvNodeGroup` object (see *Node Groups* on page 172) to the constraint in the following way:

```
layout.addConstraint(new IlvLevelRangeConstraint(nodeGroup, 5, 7));
```

Level Index Parameter

The level index is a special case of a level range constraint (see *Level Range Constraints* on page 173). It forces the node to one particular level. For your convenience, you can specify the level index of a node directly by the following method:

```
void setSpecNodeLevelIndex(Object node, int index)
```

You pass a single node as the first argument (not a node group). The default index value is `-1`. If the default value is used, or if a node is set to a negative level index, the level index is considered to be unspecified. In this case the layout algorithm automatically calculates an appropriate level index during the leveling phase of the algorithm.

To obtain the specified level index for a node, use the following method:

```
int getSpecNodeLevelIndex(Object node)
```

However, this method returns the value that was set by `setSpecNodeLevelIndex`. If the level index was specified by allocating a corresponding level range constraint that has the same meaning, `getSpecNodeLevelIndex` still returns `-1`.

> ***Warning:*** *Using arbitrarily large level indices is not recommended. For instance, if you set the level index of a node to* `100000`*, the layout algorithm creates 100,000 levels even if the graph has far fewer nodes. This causes the layout algorithm to become unnecessarily slow.*

Same Level Constraints

If you want to force several nodes to the same level with fixed index, you can set the level index parameter of these nodes accordingly (see *Level Index Parameter* on page 174) or use a level range constraint (see *Level Range Constraints* on page 173). However, if you want to force several nodes to the same level *without* forcing them to a specific level index, you cannot use these mechanisms. You must use a same level constraint:

```
layout.addConstraint(new IlvSameLevelConstraint(node1, node2));
```

This forces `node1` and `node2` to be placed into the same level, but it does not constrain them to any particular level.

Figure 4.82 illustrates the placement of nodes on the same level.

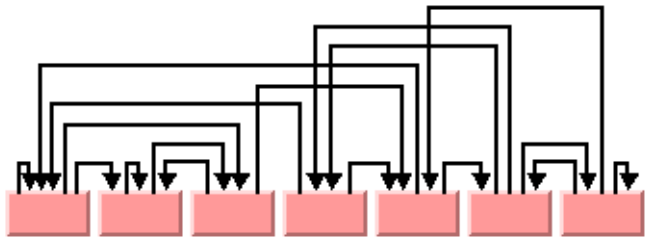


Figure 4.82 *All Nodes Fixed at Same Level*

Group Spread Constraint

An alternative way to force a group of nodes to the same level is by specifying a group spread constraint with a spread size of zero (0). In general, the group spread constraint forces a group of nodes to *k+1* subsequent levels. The number *k* is the spread size. It does not select the lowest or highest level index of the group, but only requires that the nodes be placed no more than *k* levels apart. Hence, if *k=0*, all nodes of the group are placed at the same level.

The following example illustrates the general group spread constraint:

```
IlvNodeGroup nodeGroup = new IlvNodeGroup();
nodeGroup.add(nodeA);
nodeGroup.add(nodeB);
nodeGroup.add(nodeC);
layout.addConstraint(new IlvGroupSpreadConstraint(nodeGroup, 2));
```

The constraint is satisfied if the highest level index for `nodeA`, `nodeB`, and `nodeC` is no more than two levels apart from the smallest level index of the nodes. For instance, the constraint is satisfied if the level indices for `nodeA`, `nodeB`, and `nodeC` are 1, 2, 3; or if they are 7, 8, 9; or if they are 16, 14, 15. The constraint is also satisfied if all three nodes are placed at level 5, or if two of the nodes are placed at level 15 and the third node at level 13. The constraint is not satisfied if the level indices for `nodeA`, `nodeB`, and `nodeC` are 3, 5, 6, because in this case the highest index (6) is more than two levels away from the lowest index (3).

Relative Level Constraint

You can force a node into a higher level than another node. If the flow direction is towards the bottom, level 0 is topmost in the drawing. In this layout you can specify by relative level constraints that a node be above or below another node. If the flow direction is towards the right, level 0 is leftmost in the drawing. Here you can specify by relative level constraints that a node be left or right of another node. For example:

```
layout.addConstraint(
    new IlvRelativeLevelConstraint(nodeA, nodeB, priority));
```

This forces `nodeA` to be placed at a level with a smaller index than `nodeB`. Since relative level constraints compete with each other, you must specify the priority of the constraint. In

fact, links also impose constraints on the system, and the link priority has the same impact as the constraint priority. A link with priority 10 forces its (usually) source node (unless ports are specified) into a lower level than its target node. To force the source node into a higher level than the target node, you need to create a constraint with a higher priority than the link. For instance, to ensure that the constraints are satisfied even if there are many links, you can use link priorities between 0 and 10 and constraint priorities between 1000 and 10,000.

You can also create a relative level constraint between groups of nodes:

```
layout.addConstraint(
    new IlvRelativeLevelConstraint(nodeGroup1, nodeGroup2, priority));
```

Position Index Parameter

In Step 2 of the layout algorithm (the crossing reduction phase), the nodes are ordered within the levels. All nodes that belong to the same level get a position index starting from 0. For instance, when the flow direction is to the bottom, the node with the position index 0 is placed in the leftmost position within its level. The nodes with a larger position index are placed farther to the right than the nodes with a smaller position index in the same level. The nodes of different levels are independent. The node of the first level with the position index 0 is to the left of the node of the first level with the position index 1, but not necessarily to the left of a node of another level with position index 0. Note that long links crossing a level also obtain a position index (see Figure 4.59 on page 139). The layout algorithm calculates these position indices automatically.

You can affect how the nodes are positioned within each level by specifying the position index of some nodes. The nodes are placed at the specified position within their level.

To specify the position index of a node, use the following method:

```
void setSpecNodePositionIndex(Object node, int index)
```

The default value is `-1`. If the default value is used, if a node is set to a negative position index, or if a node is set to a position index that is larger than the number of nodes of its level, the layout automatically calculates an appropriate position index during the crossing reduction step.

To obtain the current position index of a node, use the following method:

```
int getSpecNodePositionIndex(Object node)
```

Relative Position Constraints

Working with absolute node position indices is inconvenient in certain situations. For instance, if two nodes belong to the same level, you may want to force one node to a position with a lower index than the other node without fixing the absolute positions of the nodes. You can achieve this by using a relative position constraint:

```
layout.addConstraint(
    new IlvRelativePositionConstraint(nodeA, nodeB, priority));
```

This forces nodeA to a lower position than nodeB. If the flow direction is towards the bottom, the nodes are in horizontal levels; hence the constraint means that nodeA is placed at the left side of nodeB. If the flow direction is towards the right, the nodes are in vertical levels; hence the constraint means that nodeA is placed below nodeB.

The relative position constraint has an effect only if both nodes actually belong to the same level. In order to achieve this, you can, for instance, use a same level constraint in addition. There is no way to influence the relative position of nodes that belong to different levels.

Similar to the relative level constraint, the relative position constraint can be applied to node groups. These constraints also have priorities that indicate which constraints dominate if a constraint conflict occurs. The higher the priority, the more likely the constraint is satisfied when resolving constraint conflicts.

Side-By-Side Constraints

The relative position constraint (see *Relative Position Constraints* on page 176) forces a specific order upon the nodes of a level, but it does not specify which nodes are directly neighbored. For instance, a relative position constraint may force nodeA to be placed somewhere at a lower position than nodeB, but there may be many nodes between nodeA and nodeB.

To force nodes to be directly neighbored, use the side-by-side constraint on a group of type IlvNodeGroup (see *Node Groups* on page 172):

```
layout.addConstraint(
    new IlvSideBySideConstraint(nodeGroup, priority));
```

If the node group consists of just two nodes, it forces the two nodes to be placed side by side. However, it does not specify which node is at the lower node position and which node is at the higher node position. If the group consists of more than two nodes, it forces the nodes to be placed at consecutive positions such that all nodes are clustered together. A node that does not belong to the group cannot be placed between the nodes of the group.

For instance, assume that the group contains the three nodes A, B, C. The constraint is satisfied if the position indices of A, B, and C are 3, 4, 5 or 9, 7, 8. However, if node D is placed between A and B (say, D has position 4, A has position 3, and C has position 5), then the constraint is not satisfied because D does not belong to the same group.

The side-by-side constraint has an effect only if the nodes actually belong to the same level. To achieve this, you can, for instance, use a same level constraint in addition.

Side-by-side constraints have priorities that decide how to resolve constraint conflicts. The higher the priority, the more likely the constraint is satisfied.

You can use side-by-side constraints to create nested clusters. For example:

```
IlvNodeGroup group1 = new IlvNodeGroup();
group1.add(nodeA);
group1.add(nodeB);
group1.add(nodeC);
```

```
group1.add(nodeD);
layout.addConstraint(
    new IlvSideBySideConstraint(group1, 10.0f));
IlvNodeGroup group2 = new IlvNodeGroup();
group2.add(nodeB);
group2.add(nodeC);
layout.addConstraint(
    new IlvSideBySideConstraint(group2, 10.0f));
```

The first constraint specifies that `nodeA`, `nodeB`, `nodeC`, and `nodeD` must be clustered. The second constraint specifies that `nodeB` and `nodeC` are clustered inside the larger cluster. This means that no other node can be placed between the four nodes and, furthermore, neither `nodeA` nor `nodeD` can be placed between `nodeB` and `nodeC`. Figure 4.83 sketches four solutions that satisfy both constraints.

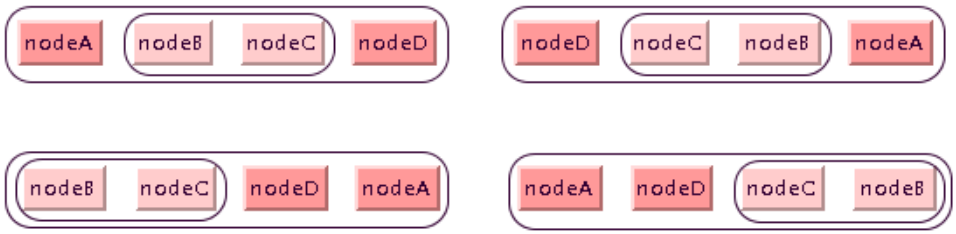


Figure 4.83 *Sketch of Solutions for Side-By-Side Constraints*

Extremity Constraints

To force a node to the first level, you can specify:

```
layout.setSpecNodeLevelIndex(node, 0);
```

However, you cannot specify a level index for the last level because it is unknown at the beginning of layout how many levels will be created. It is unwise to specify:

```
layout.setSpecNodeLevelIndex(node, java.lang.Integer.MAX_VALUE);
```

because this will create many empty levels between the levels actually used and the last one. Even though these empty levels are removed in postprocessing steps, this influences the speed and quality of the layout. (In fact, the algorithm will run out of memory if you set the specified level index unreasonably high.)

By using constraints you can achieve the same effect more efficiently. To force a node to the first level, you can specify:

```
layout.addConstraint(
    new IlvExtremityConstraint(node, IlvHierarchicalLayout.NORTH));
```

To force a node to the last level, you can specify:

```
layout.addConstraint(
    new IlvExtremityConstraint(node, IlvHierarchicalLayout.SOUTH));
```

With compass directions as a convenient reference (see *Port Sides Parameter* on page 153), the first level indicates the north pole and the last level indicates the south pole. You can also specify extremity constraints for the east and west sides:

```
layout.addConstraint(
    new IlvExtremityConstraint(node1, IlvHierarchicalLayout.EAST));
layout.addConstraint(
    new IlvExtremityConstraint(node2, IlvHierarchicalLayout.WEST));
```

The west extremity constraint forces the node to the lowest position index within its level, and the east extremity constraint forces the node to the highest position index within its level. The position indices specify the relative position within the level. For instance, a node with west extremity constraint will be the leftmost node within its level, if the flow direction is towards the bottom. However, this does not affect other levels; there may be a node in another level that is still placed farther to the left.

Figure 4.84 illustrates some extremity constraints.

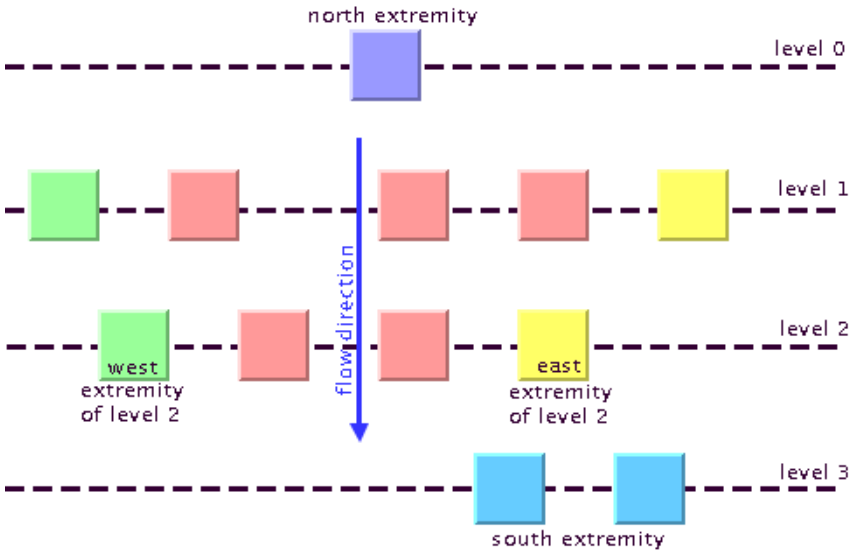


Figure 4.84 *Sketch of Extremity Constraints*

Swim Lane Constraint

Swim lanes are rectangular areas orthogonal to the levels.

- If the link flow direction is towards the bottom or top, the levels are horizontal rows and the swim lanes are vertical columns.
- If the flow direction is towards the left or right, the levels are vertical columns and the swim lanes are horizontal rows.

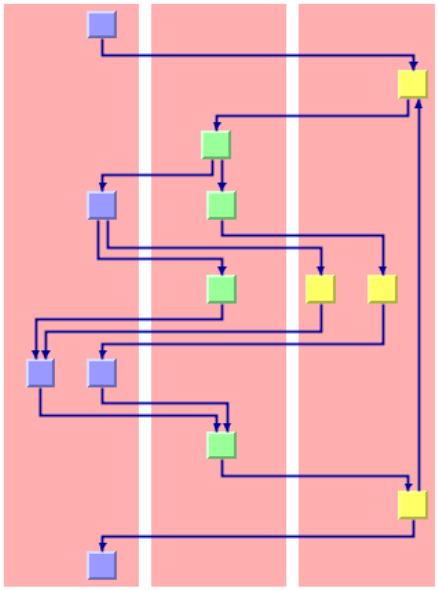
Swim lanes can be used if the nodes are partitioned into groups, to indicate which nodes belong to a certain group. The nodes of the same swim lane are placed so that it is possible to draw a surrounding rectangle around them. Swim lanes allow you to organize the graph in a table-like manner. For instance, you may have a workflow diagram where nodes represent actions; then the swim lanes could represent the departments that perform these actions. Each node can belong to only one swim lane.

In order to specify that a group of nodes belong to the same swim lane, call:

```
layout.addConstraint(new IlvSwimLaneConstraint(new IlvNodeGroup(nodeVector)));
```

All nodes of the node vector will be placed in the same swim lane rectangle. If a graph has many swim lane rectangles, the relative order of these swim lanes is determined automatically. The size of the swim lane rectangle depends on the nodes that belong to the swim lane. However, you can specify the relative order, relative size, and the margins of the swim lane as well by using the constructor:

```
public IlvSwimLaneConstraint(IlvNodeGroup group,
                             float relativeSize,
                             int positionIndex,
                             float minMargin)
```



The red background rectangle indicates where the swim lane is.

Figure 4.85 *Swim Lanes*

The relative size indicates how large this swim lane is compared to the other swim lanes. Assume that the flow direction is towards the bottom. In this case, the relative size indicates the width of the swim lane. All swim lanes with the same relative size will have the same width. A swim lane with a relative size that is twice the value of another swim lane will have twice the width of the other swim lane. The actual number of this parameter does not matter, only how large the value is compared to the other swim lanes. If you do not want to restrict the size of the swim lane, set the value to 0. In this case, the width of the swim lane will be independent of the other swim lanes.

The minimal margin is the margin of the swim lane in absolute coordinates. If the flow direction is towards the bottom, then it is the minimal horizontal distance between the leftmost or rightmost node of the swim lane and the swim lane border.

The position index indicates the order of the swim lanes. Just as nodes have position indices, the swim lanes are placed sequentially at relative positions numbered from 0 to n. In a top-down layout, the swim lane with position 0 is the leftmost swim lane, and the swim lanes with higher position indices are placed farther to the right. If the swim lanes have the position index -1, the layout algorithm determines the appropriate position automatically.

A swim lane constraint is always evaluated, even if the incremental mode is enabled. The constraint has a higher priority than the relative position constraint and the side-by-side constraint. You can specify side-by-side constraints for a group of nodes that belong to the same swim lane, but side-by-side constraints of nodes of different swim lanes are ignored. You can specify relative position constraints between nodes of the same swim lane. You can

also specify relative position constraints between one entire swim lane group and another swim lane group, which effectively orders the swim lanes. But relative position constraints are ignored if they would require breaking the swim lanes apart. The swim lane constraint dominates the specified position indices and the extremity constraints, that is, if a swim lane constraint is used, you cannot specify position indices or east/west extremity constraints for any node.

> ***Tip:*** *The automatic conflict resolution can handle conflicting constraints. However, to speed up the layout, it is recommended that you specify constraints in such a way that there are no conflicts.*

Constraint Priorities

A set of constraints may cause conflicts. This means that not all of the constraints can be satisfied at the same time. For instance, it is impossible to force two nodes into the same level by an `IlvSameLevelConstraint` while at the same time forcing one of the nodes to a higher level than the other node by an `IlvRelativeLevelConstraint`. In this case, one of the two constraints must be ignored during layout.

In general, constraint conflicts are resolved by ignoring the constraints with the lowest priority while the constraints with the highest priority get satisfied. The following rules explain the constraint priorities in detail.

- The constraints that influence into which level a node is placed are applied before the constraints that influence the position of the node within a level.
- The `IlvExtremityConstraint` is translated into a sequence of constraints with high priority. For instance, the extremity constraint with the south side is translated into several same level constraints and several relative level constraints.
- The `IlvSameLevelConstraint` and the `IlvGroupSpreadConstraint` have the highest priority. They are never in conflict with each other. They dominate all other constraints, even the specified level index.
- The `IlvLevelRangeConstraint` (and the direct level index specification) has the second highest priority. If two nodes are forced to the same level but have disjoint specified level ranges, then the level range is ignored. In the following example:

  ```
  layout.addConstraint(new IlvSameLevelConstraint(node1, node2));
  layout.setSpecNodeLevelIndex(node1, 5);
  layout.setSpecNodeLevelIndex(node2, 10);
  ```

  both `node1` and `node2` will be placed at level 5. The conflicting specification: `layout.setSpecNodeLevelIndex(node2, 10)` is ignored.
- The `IlvRelativeLevelConstraint` is dominated by the same level constraint, by the level range constraint, and by the direct specification of level indices. If several relative level constraints conflict each other, the one with the highest specified priority

dominates. However, note that all links are implicitly considered relative level constraints as well. If links with high priority force a node to a certain level, then a relative level constraint with lower priority will be ignored.

- The `IlvSwimLaneConstraint` is always evaluated, even if the incremental mode is enabled. The constraint has a higher priority than the relative position constraint and the side-by-side constraint. You can specify side-by-side constraints for a group of nodes that belong to the same swim lane, but side-by-side constraints of nodes of different swim lanes are ignored. You can specify relative position constraints between nodes of the same swim lane. You can also specify relative position constraints between one entire swim lane group and another swim lane group, which effectively orders the swim lanes. But relative position constraints are ignored if they would require breaking the swim lanes apart. The swim lane constraint dominates the specified position indices and the extremity constraints, that is, if a swim lane constraint is used, you cannot specify position indices or east/west extremity constraints for any node.
- The `IlvSideBySideConstraint` is evaluated only if the corresponding nodes belong to the same level. Typically you will use a same level constraint to force the nodes to the same level, and then a side-by-side constraint to force the nodes to a certain ordering. The side-by-side constraints dominate the relative position constraints. If several side-by-side constraints are conflicting, the one with the highest specified priority dominates the other constraints.
- The `IlvRelativePositionConstraint` is also evaluated only if the corresponding nodes belong to the same level. It is dominated by the side-by-side constraint; however, conflicts with side-by-side constraints are rare. If several relative position constraints are conflicting, the one with the highest specified priority dominates the other constraints.

For Experts: Constraint Validation

Constraints may become invalid. For instance, if you add a constraint that node `A` must be to the left side of node `B`, but you remove `A` from the graph, then this constraint becomes invalid. It simply does not make sense any more, even though it does not conflict with any other constraint. The layout instance automatically removes invalid constraints from time to time because they are a waste of memory. The validation check is done during layout. Forcing a validation check is normally not needed but if you want to perform it, call:

```
layout.validateConstraints();
```

This removes all invalid constraints from the Hierarchical Layout and cleans up the memory. The constraint validation does not check which constraints have conflicts. The main effect of the validation is that the constraint system uses less memory afterwards.

> ***Note:*** *A constraint is valid if it is meaningful. Two valid constraints are conflicting if the system cannot satisfy them both at the same time. Invalid constraints cannot be conflicting because they are meaningless.*
>
> *Hence, constraint validation and constraint resolution are different phases. Constraint validation performs a quick local test. It removes invalid constraints from the layout instance completely. It does not affect conflicting constraints.*
>
> *Constraint resolution checks whether a set of valid constraints are in conflict with each other. Thus, constraint resolution is a complex process on a network of multiple related constraints. Constraint resolution decides which constraints can be solved and which cannot. But the constraint resolution does not remove conflicting constraints from the layout instance, it just delivers a solution that may ignore some constraints.*

For Experts: More Indices

The Hierarchical Layout allows you to specify the level index and the position index of a node. How these indices are used depends on the graph topology and the additional constraints. It is also possible to obtain the indices of nodes that were calculated during layout.

Calculated Level Index

The layout algorithm allows you to access the level index that was calculated for a node by a previous layout. To do this, use the following method:

```
int getCalcNodeLevelIndex(Object node)
```

If the node was never laid out, this method returns `-1`. Otherwise, it returns the previous level index of the node.

The method can be used to specify the level index for the next layout in the following way:

```
int index = layout.getCalcNodeLevelIndex(node);
layout.setSpecNodeLevelIndex(node, index);
```

When this is done, it ensures that the node is placed at the same level as in the previous layout.

If the graph is detached from the layout algorithm, the calculated level index of a node is set back to `-1`.

Note: *You should be aware of the difference between the methods* `getCalcNodeLevelIndex` *and* `getSpecNodeLevelIndex`*. The first one returns the level index calculated by the previous layout. The second one returns the specified level index, even if there was no previous layout.*

For instance, consider two nodes A and B. Node A has no specified level index and node B has a specified level index 5. Before the first layout, the method `getCalcNodeLevelIndex` *returns -1 for both nodes because the levels have not been calculated yet. However,* `getSpecNodeLevelIndex` *returns -1 for A and 5 for B. After the first layout, node A may be placed at level 4. Now,* `getCalcNodeLevelIndex` *returns 4 for node A and 5 for node B and* `getSpecNodeLevelIndex` *still returns -1 for A and 5 for B.*

Calculated Position Index

The layout algorithm allows you to access the position index within a level that was calculated for a node by a previous layout. To do this, use the following method:

```
int getCalcNodePositionIndex(Object node)
```

If the node was never laid out, this method returns -1. Otherwise, it returns the previous position index of the node within its level.

To ensure that the node is placed at the same level at the same relative position as in the previous layout, use the following:

```
layout.setSpecNodeLevelIndex(node,
    layout.getCalcNodeLevelIndex(node));
layout.setSpecNodePositionIndex(node,
    layout.getCalcNodePositionIndex(node));
```

This example code works only if the generic connected component layout is disabled and the port sides `EAST` or `WEST` are not used in the layout.

If the graph is detached from the layout algorithm, the calculated position index of a node is set back to -1.

Note: *You should be aware of the difference between the methods* `getCalcNodePositionIndex` *and* `getSpecNodePositionIndex`*. The first one returns the position index calculated by the previous layout and* `-1` *if there was no previous layout. The second one returns the specified position index even if there was no previous layout. This behavior is similar to the behavior of the specified and calculated level index (see section Calculated Level Index on page 184).*

Link Layout

In this section, you will learn about the *Link Layout* algorithm from the ILOG JViews Graph Layout module (class `IlvLinkLayout` from the package `ilog.views.graphlayout.link`).

Samples

Here are sample drawings produced with the Link Layout:

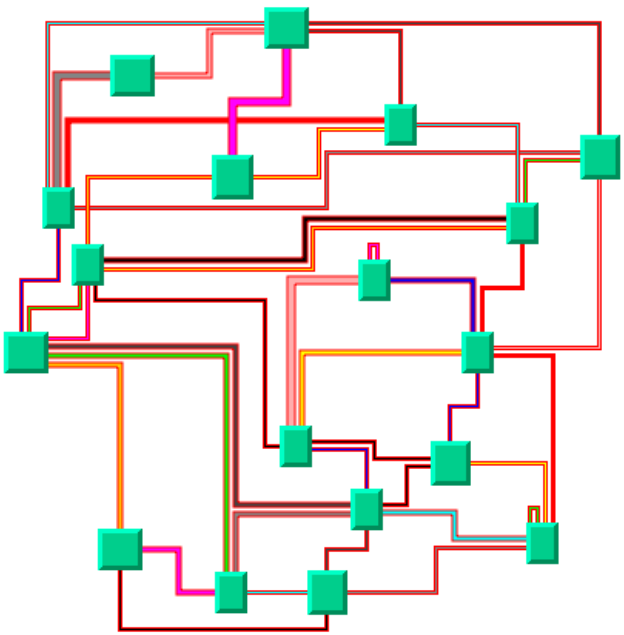


Figure 4.86 *Link Layout in Short Link Mode with Orthogonal Links*

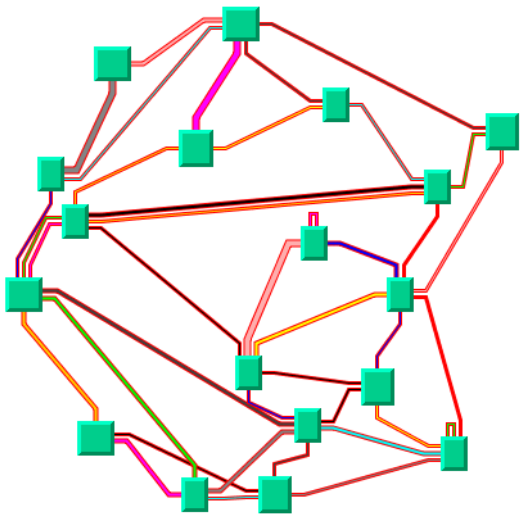


Figure 4.87 *The Same Graph in Short Link Mode with Direct Links*

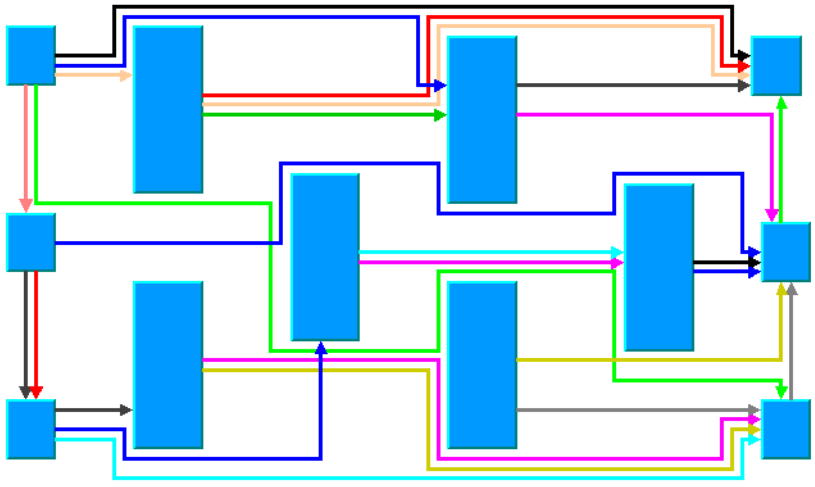


Figure 4.88 *Link Layout in Long Link Mode with Orthogonal Links*

What Types of Graphs?

Any type of graph where nodes are fixed and links need to be routed:

- Connected and disconnected graphs
- Planar and nonplanar graphs
- Nested graphs with intergraph links

Application Domains

Application domains of the Link Layout include:

- Electrical engineering (circuit block diagrams)
- Industrial engineering (schematic design diagrams, equipment/resource control charts)
- Business processing (entity relation diagrams)
- Software management/software (re-)engineering (data inspector diagrams)
- Database and knowledge engineering (sociology, genealogy)
- CASE tools (design diagrams)

Features

- Reshapes the links of a graph in either an orthogonal or a direct style, without moving the nodes. Orthogonal and direct style links can be combined in the same layout.
- Allows you to specify which side of the node (top, bottom, left, or right) a link can be connected to, or to preserve the existing connection points of the links.
- Supports self-links (that is, links with the same origin and destination node).
- Supports multiple links (that is, more than one link between the same origin and destination nodes).
- Allows you to specify pinned (fixed) links that the layout algorithm cannot reshape.
- Supports intergraph links of nested graphs. An intergraph link is a link whose end nodes belong to different subgraphs of a nested graph.
- Supports an incremental mode: If new links are added to a drawing, the next layout takes the shapes of the old links into account.
- Two layout modes: *short links* with a limited number of bends or *long links* with unlimited number of bends.

  Short Link Mode

  - Links are placed freely in the space.
  - Link-to-link and link-to-node crossings are reduced, if this is possible with link shapes that have a maximum of 4 bends.

- Links of different width are supported.
- Link bundles between the same pair of nodes are supported. Optionally, the algorithm can ensure that multiple links are bundled together by giving them parallel shapes.
- Automatically arranges the final segments of the links (the segments near the origin or destination node) to obtain a bundle of parallel links.
- Provides two optional shapes for the self-links.
- Very fast algorithm with low memory footprint.

Long Link Mode

- Links are placed on a grid.
- Link-to-node crossings of orthogonal links are avoided, even if this introduces many bends.
- Orthogonal link segments do not overlap.
- Does not bundle the final segments. Instead, it distributes the links on the border of each end node according to which border has more free space.
- Fast algorithm: speed and memory footprint depend on the grid spacing.

Limitations

- Since the Link Layout algorithm reshapes the links, it works preferably with links of type `IlvPolylineLinkImage` and link connectors of type `IlvRelativeLinkConnector`.
- When routing intergraph links, the incremental mode cannot be used. Due to the complexity of intergraph link routing, more crossings and overlappings may occur than when routing normal links.
- In short link mode, crossings and overlapping of links with other links and nodes cannot always be avoided because the algorithm uses link shapes with a limited number of bends. This happens in particular when there are many obstacles between the end points of a link.
- In long link mode, link crossings cannot always be avoided. Segment overlappings of orthogonal links are always avoided unless there is no free space remaining on the border of the end nodes. Any overlapping of nodes and links is always avoided unless one end nodes is inside an enclave. An enclave is an area that is surrounded by other nodes such that the area cannot be reached from the other end node. (See Figure 4.103 on page 215.)
- In long link mode, segment overlapping or overlapping between nodes and links cannot always be avoided if the direct link style is used.
- The long link mode is slower and uses more memory if the grid spacing is very small.

Brief Description of the Algorithm

The Link Layout algorithm utilizes two sublayout classes:

- `IlvShortLinkLayout` in short link mode.
- `IlvLongLinkLayout` in long link mode.

Both implement different strategies to find the link shapes.

Short Link Layout Algorithm

The Short Link Layout algorithm is based on a combinatorial optimization that chooses the "optimal" shape of the links in order to minimize a cost function. This cost function is proportional to the number of link-to-link and link-to-node crossings.

For efficiency reasons, the basic shape of each link is chosen from a set of predefined shapes. These shapes are different for each link style option. For the orthogonal link style, the links are reshaped to a polygonal line of up to five alternating horizontal and vertical segments (see Figure 4.86 on page 186). For the direct link style, the links are reshaped to a polygonal line composed of three segments: a straight-line segment that starts and ends with small horizontal or vertical segments (see Figure 4.87 on page 187).

The shape of a link also depends on the relative position of the origin and destination nodes. For instance, when two nodes are very close or they overlap, the shape of the link is chosen to provide the best visibility of the link.

The exact shape of a link is computed taking into account additional constraints. The layout algorithm tries to do the following:

- Minimize the number of crossings between the links incident to a given side of a node.
- Space the final segments of the links incident to a given side of a node equally on the node border.

Long Link Layout Algorithm

The Long Link Layout algorithm first treats each link individually. For each link, it first calculates the connection points at the end nodes that are on the grid and orders them according to a penalty value. Connection points on used grid points have a very high penalty and, therefore, are very unlikely to be used.

For the orthogonal links (see Figure 4.88 on page 187), the Long Link Layout algorithm then uses a grid traversal to search a route over free grid points from the start connection point to the end connection point. Therefore, in contrast to the short link mode, orthogonal links can have any shape with a large number of bends if this is necessary to bypass obstacle nodes to avoid overlappings. For the direct links (see Figure 4.87 on page 187), it shortens the search by using a direct segment between the connection points.

After all links are placed, a crossing reduction phase examines pairs of links and eliminates link crossings by exchanging parts of the routes between both links.

The Long Link Layout algorithm relies on the fact that links fit to the grid spacing and parts of the routes between different links can be exchanged. Therefore, the Long Link Layout algorithm does not take the link width into account because it would be too difficult to find the parts of two links that can be exchanged. It is recommended to set the grid spacing larger than the largest link width.

Code Sample

Below is a code sample using the `IlvLinkLayout` class:

```
...
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.link.*;
 ...
IlvGrapher grapher = new IlvGrapher();
IlvManagerView view = new IlvManagerView(grapher);

 ... /* Fill in the grapher with nodes and links here */

IlvLinkLayout layout = new IlvLinkLayout();
layout.attach(grapher);

/* Specify the layout mode */
layout.setLayoutMode(IlvLinkLayout.SHORT_LINKS);

try {
        IlvGraphLayoutReport layoutReport = layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Generic Features and Parameters

The `IlvLinkLayout` class supports the following generic parameters as defined in the class `IlvGraphLayout`:

- *Allowed Time*
- *Animation*
- *Automatic Layout*

- *Preserve Fixed Links*
- *Save Parameters to Named Properties*
- *Stop Immediately*

The following comments describe the particular way in which these parameters are used by this subclass.

Allowed Time

The layout algorithm stops if the allowed time setting has elapsed. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Allowed Time* on page 22.) If the layout stops early because the allowed time has elapsed, some links may not be routed in the best possible way. The result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Animation

The layout algorithm can show the temporary positions of the links during the routing in an animated way. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Animation* on page 22.)

Note: *If this option is enabled, the layout of large graphs can be very time consumming.*

Automatic Layout

The Link Layout routes the links so that they bypass the nodes and cross each other as few times as possible. It does not position any nodes. However, if the user moves, adds, or resizes nodes, or adds or removes links, the Link Layout drawing usually becomes invalid; that is, the links no longer look orthogonal, overlap the moved nodes, or cross other links.

Using the automatic layout feature of the `IlvGraphLayout` class, the layout is reperformed whenever a change of the graph occurs. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Automatic Layout* on page 23.)

Preserve Fixed Links

The layout algorithm does not reshape the links that are specified as fixed. (See *Preserve Fixed Links* on page 32.) The fixed links are taken into account when computing the optimal layout of the nonfixed links.

Save Parameters to Named Properties

The layout algorithm can save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Save Parameters to Named Properties* on page 34 and *Saving Layout Parameters and Preferred Layouts* on page 315.)

Stop Immediately

The layout algorithm stops if the method `stopImmediately` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop Immediately* on page 35.) If the layout stops early, some links may not be routed in the best possible way. The result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific Parameters for Both Layout Modes

The following parameters are specific to the `IlvLinkLayout` class.

Layout Mode

The Link Layout algorithm has two layout modes. To select a layout mode, use the method

```
void setLayoutMode(int mode);
```

To obtain the current layout mode, call:

```
int getLayoutMode();
```

The valid values for `mode` are:

- `IlvLinkLayout.SHORT_LINKS` (the default)
- `IlvLinkLayout.LONG_LINKS`

Figure 4.89 shows a small sample graph in short and long link mode. The short link mode bundles the links very well. However, due to the bundling, some red links appear to be unconnected to the green nodes. Furthermore, the algorithm cannot find a route for the long red links without overlapping some nodes or without overlapping the green link. The long link mode works on a grid. It is specialized for long links and avoids overlapping any nodes or link segments. It can connect to the green nodes by choosing connection points on different sides of the end nodes. This advantage, however, is paid for by a less regular structure that does not bundle the links and a larger number of link crossings.

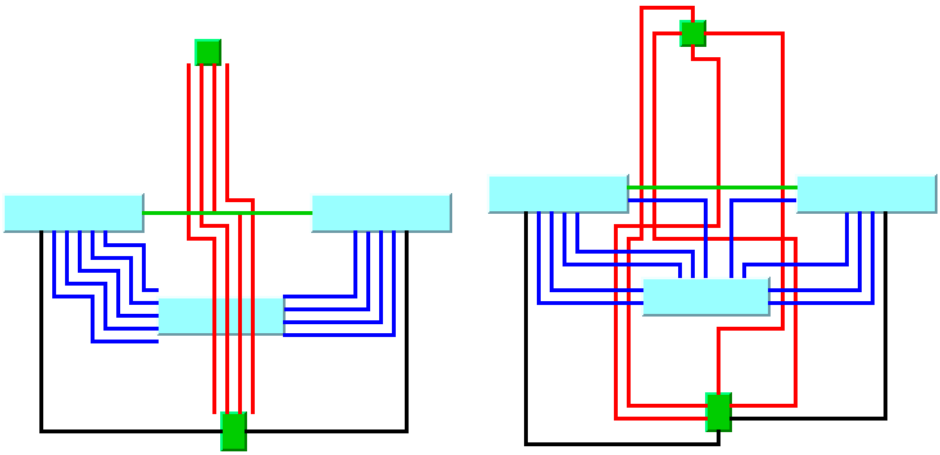


Figure 4.89 *Short and Long Link Modes with Orthogonal Links*

Choosing the Appropriate Layout Mode

The short link mode should be used if any of the following conditions apply:

- The majority of links is short and it is not fatal if long links overlap obstacles.
- The link routes must be placed freely and cannot be restricted to a grid.
- It is important to limit the number of bends.

The long link mode should be used if any of the following conditions apply:

- Many links are long and it is important that long links do not overlap obstacles.
- There is a preferred routing because the nodes are already placed on the grid.
- It is important to have a guaranteed a minimal distance between link segments.
- An increasing number of bends is acceptable if it avoids any overlappings.

Figure 4.90 shows how the long link mode can be used to find an orthogonal route without overlappings in a labyrinth of node obstacles.

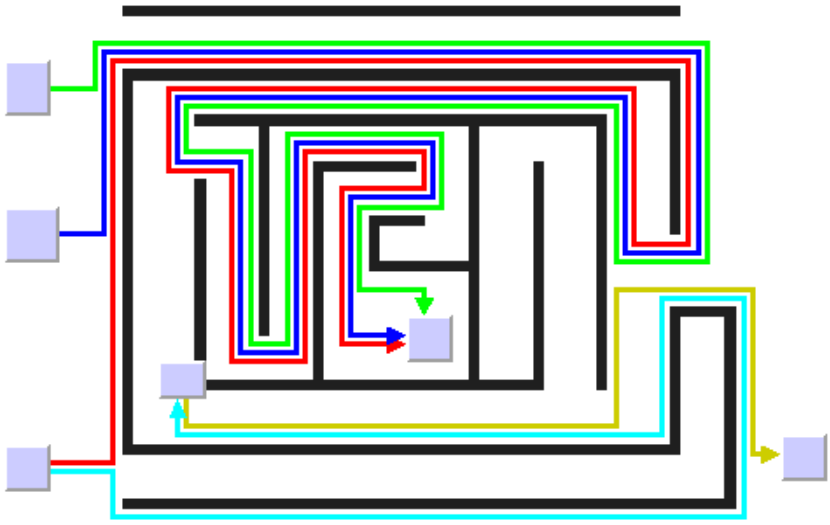


Figure 4.90 *Labyrinth Routing with the Long Link Mode*

Link Style

The layout algorithm provides two link styles. You can set the link style globally, in which case all links have the same kind of shape, or locally on each link, in which case different link shapes occur in the same drawing.

> ***Note:*** *The layout algorithm may raise an* `IlvInappropriateLinkException` *if layout is performed on an* `IlvGrapher`*, but links are neither a subclass of* `IlvPolylineLinkImage` *nor of* `IlvSplineLinkImage`*, or if connectors are not a subclass of* `IlvRelativeLinkConnector`*. You can use the methods* `EnsureAppropriateLinkTypes`*,* `EnsureAppropriateLinkConnectors` *or* `EnsureAppropriateLinks` *defined in* `IlvGraphLayoutUtil` *to automatically replace inappropriate links or link connectors before layout or when the* `IlvInappropriateLinkException` *is caught. For details on these methods, see the ILOG JViews Graph Layout Reference Manual. For details on the graph model, see Using the Graph Model on page 297.*

Global Link Style

To set the global link style, the following method is provided:

```
void setGlobalLinkStyle(int style);
```

The valid values for `style` are:

- `IlvLinkLayout.ORTHOGONAL_STYLE` (the default)

The links are reshaped in an orthogonal form (alternating horizontal and vertical segments). See Figure 4.86 on page 186 and Figure 4.88 on page 187 as examples.

- `IlvLinkLayout.DIRECT_STYLE`

  The links are reshaped to a polygonal line composed of three segments: a straight-line segment that starts and ends with a small horizontal or vertical segment. See Figure 4.87 on page 187 as an example.

- `IlvLinkLayout.MIXED_STYLE`

  Each link can have a different link style. The style of each individual link can be set to have different link shapes occurring on the same graph.

To obtain the current choice, use the following method:

```
int getGlobalLinkStyle();
```

Individual Link Style

All links have the same style of shape unless the global link style is `IlvLinkLayout.MIXED_STYLE`. Only when the global link style is set to `MIXED_STYLE` can each link have an individual link style.

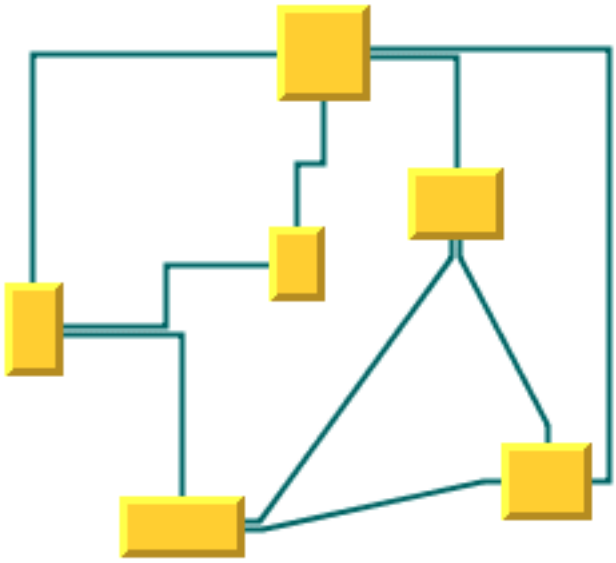


Figure 4.91 *Different Link Styles Mixed in the Same Drawing (Short Link Mode)*

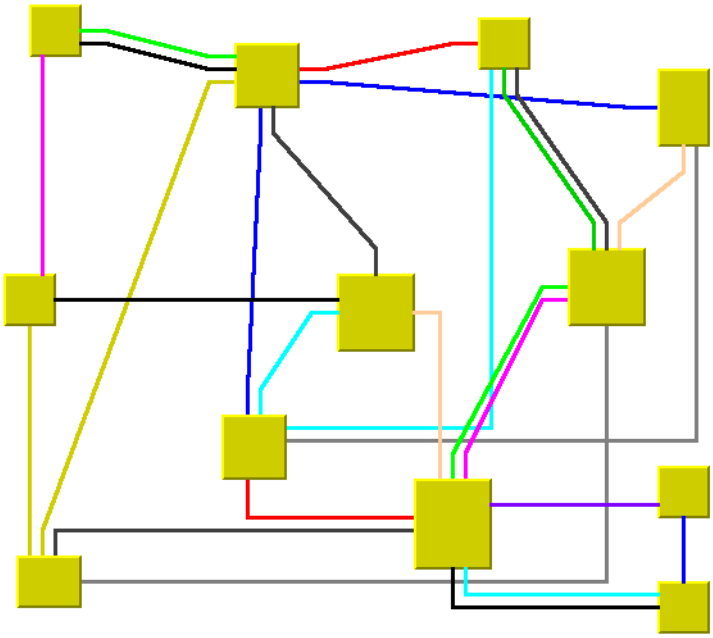


Figure 4.92 *Different Link Styles Mixed in the Same Drawing (Long Link Mode)*

To set and retrieve the style of an individual link, use the following methods:

```
void setLinkStyle(Object link, int style);
```

```
int getLinkStyle(Object link);
```

The valid values for `style` are:

- `IlvLinkLayout.ORTHOGONAL_STYLE` (the default)
- `IlvLinkLayout.DIRECT_STYLE`
- `IlvLinkLayout.NO_RESHAPE_STYLE` (that is, the link is not reshape in any manner)

> ***Note:*** *The link style of a Link Layout graph requires links in an* `IlvGrapher` *that can be reshaped. Links of type* `IlvLinkImage`, `IlvOneLinkImage`, `IlvDoubleLinkImage`, `IlvOneSplineLinkImage`, *and* `IlvDoubleSplineLinkImage` *cannot be reshaped. You should use the class* `IlvPolylineLinkImage` *or* `IlvSplineLinkImage` *instead.*

End Points Mode

Normally, the layout algorithm is free to choose the termination points of each link. However, if fixed-link connectors are used (for instance, `IlvPinLinkConnector`), the user can specify that the current fixed termination pin of a link should be used.

The layout algorithm provides two end point modes. You can set the end point mode globally, in which case all end points have the same mode, or locally on each link, in which case different end point modes occur in the same drawing.

Global End Point Mode

To set the global end point mode, the following methods are provided:

```
void setGlobalOriginPointMode(int mode);

void setGlobalDestinationPointMode(int mode);
```

The valid values for `mode` are:

- `IlvLinkLayout.FREE_MODE` (the default)

  The layout is free to choose the appropriate position of the connection point on the origin/destination node.

- `IlvLinkLayout.FIXED_MODE`

  The layout must keep the current position of the connection point on the origin/destination node.

- `IlvLinkLayout.MIXED_MODE`

  Each link can have a different end point mode.

The connection points are automatically considered as fixed if they are connected to *grapher pins*.

To obtain the current choice, use the following methods:

```
int getGlobalOriginPointMode();

int getGlobalDestinationPointMode();
```

Individual End Point Mode

All links have the same end point mode unless the global end point mode is `IlvLinkLayout.MIXED_MODE`. Only when the global end point mode is set to `MIXED_MODE` can each link have an individual end point mode.

To set and retrieve the mode of an individual link, use the following methods:

```
void setOriginPointMode(Object link, int mode);

int getOriginPointMode(Object link);

void setDestinationPointMode(Object link, int mode);
```

```
int getDestinationPointMode(Object link);
```

The valid values for `mode` are:

- `IlvLinkLayout.FREE_MODE` (the default)
- `IlvLinkLayout.FIXED_MODE`

The connection points are automatically considered as fixed if they are connected to *grapher pins*.

> ***Note:*** *The layout algorithm may raise an* `IlvInappropriateLinkException` *if layout is performed on an* `IlvGrapher`*, but links are neither a subclass of* `IlvPolylineLinkImage` *nor of* `IlvSplineLinkImage`*, or if connectors are not a subclass of* `IlvRelativeLinkConnector`*. You can use the methods* `EnsureAppropriateLinkTypes`*,* `EnsureAppropriateLinkConnectors` *or* `EnsureAppropriateLinks` *defined in* `IlvGraphLayoutUtil` *to automatically replace inappropriate links or link connectors before layout or when the* `IlvInappropriateLinkException` *is caught. For details on these methods, see the ILOG JViews Graph Layout Reference Manual. For details on the graph model, see Using the Graph Model on page 297.*

Incremental Mode

The Link Layout algorithm normally routes all links from scratch. If the graph changes incrementally because you add or remove links or nodes, the subsequent layout may differ considerably from the previous layout. To avoid this effect and to help the user to retain a mental map of the graph, the algorithm has an incremental mode.

To enable the incremental mode, use the following call:

```
layout.setIncrementalMode(true);
```

In incremental mode, the layout tries to minimize the changes to the layout. A link is only rerouted if it is new, if a link bend moved, if its layout parameters have changed, or if a node was moved such that it overlaps the link.

In short link mode, if the next layout is incremental, the links preserve the connection side and the general shape calculated by a previous layout, except if one of their end nodes has been moved or resized.

In the long link mode, a new route is searched for the links that are no longer on the grid or that overlap with nodes. The shape and the connection side of the rerouted links can change completely. However, links that are already on the grid and do not overlap nodes or other links are not rerouted in incremental mode. It is also possible to specify which link must be rerouted by the next incremental layout even though the layout has not changed. To select an individual link to be used for incremental rerouting, use the following method:

```
void markForIncremental(Object link);
```

Intergraph Link Routing

A nested graph is a graph with nodes that are subgraphs. In a nested graph, normal links and intergraph links can occur (see *Nested Managers and Nested Graphers* in the *ILOG JViews Graphics Framework User's Manual*). Normally, both end nodes of a link belong to the same subgraph. Intergraph links are those links whose end nodes belong to different subgraphs. Intergraph links belong to the lowest common grapher in the nesting structure that contains both end nodes. Figure 4.93 shows a nested graph with blue normal links and red intergraph links.

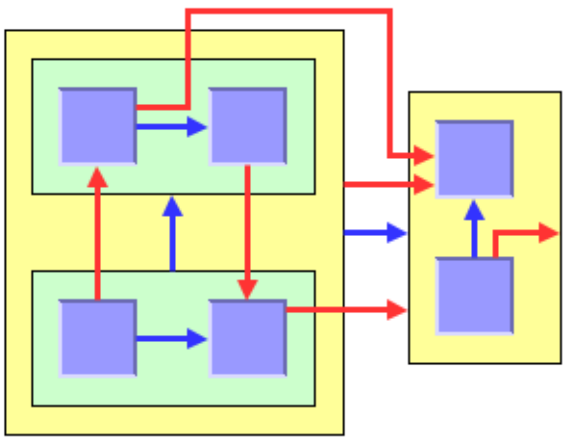


Figure 4.93 *Nested Graph With Normal Links (blue) and Intergraph Links (red)*

By default, the Link Layout routes only the normal links of a graph. But it can route intergraph links as well. To enable intergraph link routing, call:

```
layout.setInterGraphLinksMode(true);
```

If the intergraph links mode is enabled, you can select whether only the intergraph links are routed, or whether the intergraph links and the normal links are routed at the same time. If you call:

```
layout.setCombinedInterGraphLinksMode(false);
```

the next layout routes the intergraph links but does not reshape any normal links. If you call:

```
layout.setCombinedInterGraphLinksMode(true);
```

the next layout routes both the normal links and the intergraph links.

When the intergraph links mode is enabled, the layout cannot route the links incrementally (see *Incremental Mode* on page 199) and the layout animation is disabled (see *Animation* on page 22).

Notice that the layout routes only those links that belong to the attached graph. In a nested graph, each subgraph is attached to a different layout instance. Therefore, when starting a normal (nonrecursive) layout for the top-level graph of Figure 4.93 on page 200, not all links are routed that are shown in that figure, but only those links that belong to the top-level graph.

Figure 4.94 shows two situations: the yellow subgraph indicates the subgraph where the nonrecursive layout is currently applied, and color of the links indicate which links are currently routed. Depending on the intergraph links mode, the red and/or blue links are routed, but the grey links are not reshaped.

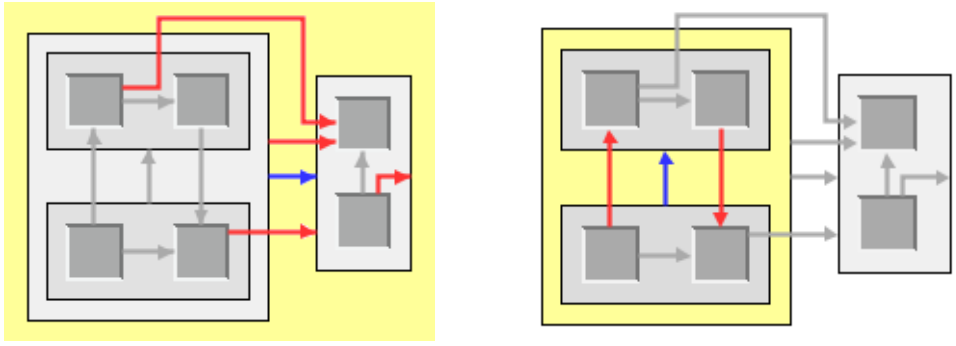


Figure 4.94 *Routed Link in a Nested Graph when Layout is Performed for the Yellow Subgraph*

In order to route *all links* of a nested graph, you need to apply the Link Layout recursively. Details of the recursive layout mechanism are explained in *Laying Out Nested Graphs* on page 306 that discusses recursive layout. For instance:

```
layout.setInterGraphLinksMode(true);
layout.performLayout(force, redraw, true);
```

routes the intergraph links recursively in all subgraphs. If you use a layout provider (a class that implements the interface `IlvLayoutProvider`), you need to set the intergraph links mode for all subgraphs explicitly:

```
IlvLayoutProvider layoutProvider = ...

// first, set the intergraph mode for all layouts
Enumeration e = graphModel.getLayouts(layoutProvider, true);
while (e.hasMoreElements()) {
    IlvGraphLayout layout = (IlvGraphLayout) e.nextElement();
    if (layout instanceof IlvLinkLayout)
        ((IlvLinkLayout) layout).setInterGraphLinksMode(true);
}
```

```
// then perform layout recursively using the provider
graphModel.performLayout(layoutProvider, force, redraw, true);
```

If you want to recursively perform the intergraph link routing in combination with a layout that places the nodes or that arranges labels, we recommend that you use an instance of the class `IlvMultipleLayout` to encapsulate the Link Layout and the other layouts, and then perform the Multiple Layout recursively all at once. For details, see *Recursive Layout* on page 265.

Spacing Parameters in Short Link Mode

Since the short link mode places the links freely in the space, only two parameters are necessary to control the spacing: the minimal distance between links and the minimal length of the final segment.

Figure 4.95 shows the spacing parameters used in the short link mode.

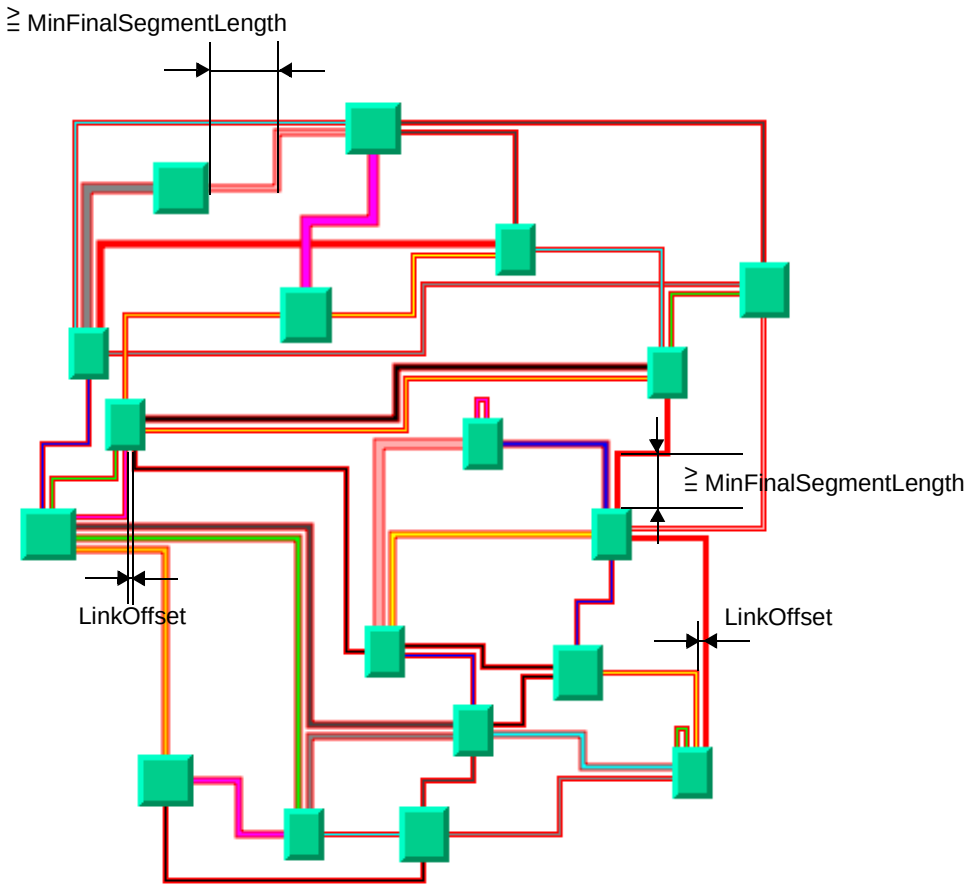


Figure 4.95 *Spacing Parameters for the Short Link Mode*

Link Offset

The layout algorithm computes the final connecting segments of the links (that is, the segments near the origin and destination nodes) in order to obtain parallel lines spaced at a user-defined distance. In short link mode, the algorithm takes into account the width of the links when computing the offset. To specify the offset, use the following method:

```
void setLinkOffset(float offset)
```

To obtain the current value, use the method:

```
float getLinkOffset()
```

The offset is measured from the border of one link to the nearest border of the other link. Therefore, if the specified offset is zero, the border of a link touches the border of its neighbor link.

Minimum Final Segment Length

You can specify a minimum value for the length of the final connecting segments of the links (that is, the segments near the origin and destination nodes) using the method:

```
void setMinFinalSegmentLength(float length)
```

To obtain the current value, use the method:

```
float getMinFinalSegmentLength()
```

The default value is 10.

Spacing Parameters in Long Link Mode

The long link mode places the links on a grid. Four parameters control the grid offsets and five parameters control the spacing of links in relation to other objects. Figure 4.96 shows the spacing parameters used in the long link mode.

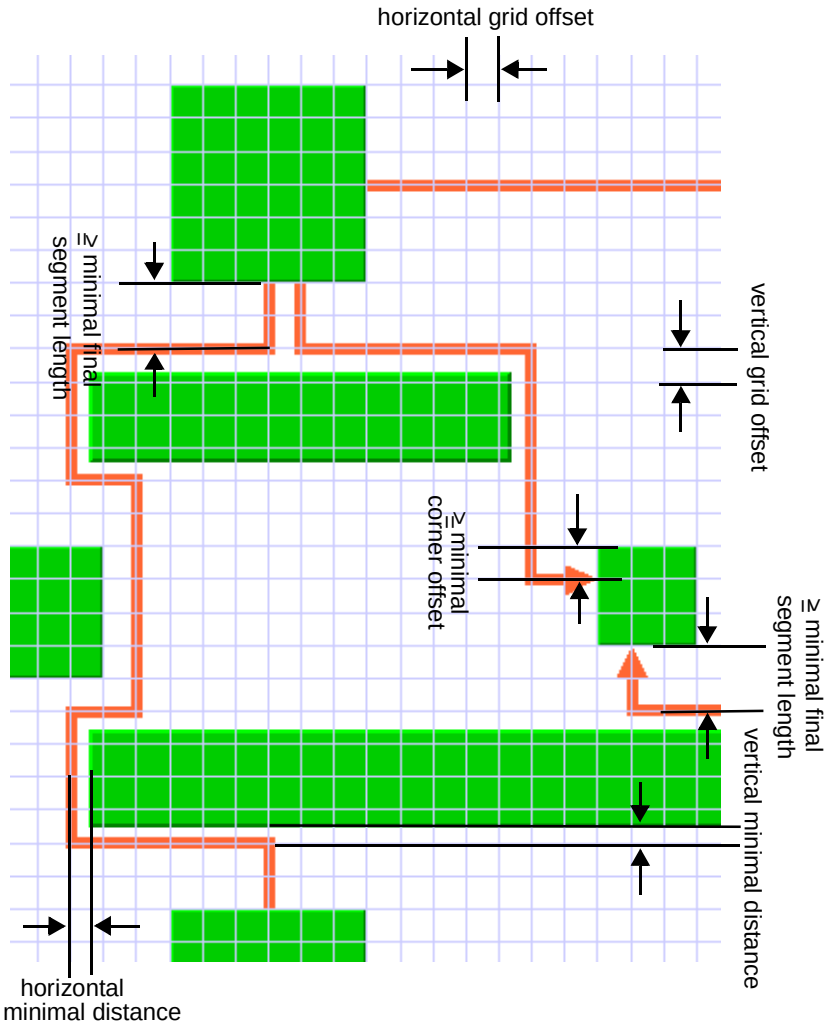


Figure 4.96 *Spacing Parameters for the Long Link Mode*

Grid Offset Parameters

The grid offset parameters control the spacing between grid lines. Links are routed such that the center of the orthogonal link segments are on the grid lines. The grid offsets should be set to a value larger than the largest link width value in order to avoid links that visually overlap.

The horizontal and vertical grid offset can be set by the following methods:

```
void setHorizontalGridOffset(float offset);

void setVerticalGridOffset(float offset);
```

They can be obtained by the corresponding methods:

```
float getHorizontalGridOffset();

float getVerticalGridOffset();
```

The grid offset is the critical parameter for the long link mode. If the grid offset is too large, there may be no grid lines between nodes even though some free space exists between the nodes. In this case, the link routings cannot use the free space. However, if the grid offset is too small, the algorithm needs a long time to traverse the grid.

Grid Base Parameters

Sometimes it is necessary to shift the whole grid by a small amount because the nodes are not aligned on the grid. For instance, to have grid lines at positions 3, 13, 23, 33, and so on, you can set the grid offset to 10 and the grid base to 3.

The grid base can be adjusted and retrieved by the following methods:

```
void setHorizontalGridBase(float coordinate);

void setVerticalGridBase(float coordinate);

float getHorizontalGridBase();

float getVerticalGridBase();
```

Minimal Distance Parameters

The minimal distance controls how closely a link can be placed to the border of a node that needs to be bypassed. If the node border is not aligned to the grid, the minimal distance specifies the next grid line close to the border that can be used. For instance, if a node covers the x-coordinates 25 to 65 on a grid with offset 10 and base 0, the next grid lines used to bypass the node would normally be at 20 and 70. If we specify a minimal distance of 8, these grid lines are too close to the node and then the grid lines at 10 and 80 would be used.

The minimal distance can be set and obtained by the following methods:

```
void setHorizontalMinOffset(float offset);

void setVerticalMinOffset(float offset);

float getHorizontalMinOffset();

float getVerticalMinOffset();
```

Minimal Node Corner Offset Parameter

The minimal corner offset is the minimal distance between a node corner and a link that connects to the node. This parameter is used to avoid having a link that connects exactly to the corner or outside the border if the node (see Figure 4.97).

The minimal corner offset can be set and obtained by:

```
void setMinNodeCornerOffset(float offset);

float getMinNodeCornerOffset();
```

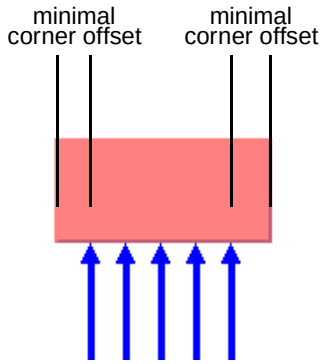


Figure 4.97 *Minimal Corner Offset*

Minimum Final Segment Length

As with the short link mode, the long link mode respects the minimum value for the length of the final connecting segments of the links. This length can be set and obtained using the methods:

```
void setMinFinalSegmentLength(float length)

float getMinFinalSegmentLength()
```

For Experts: Additional Features

The following features are available in both layout modes.

Using a Node-Side Filter

Some applications require that links are not connected to specific sides of certain nodes. The Link Layout algorithm allows you to restrict which node side a link can connect to by using a node-side filter. A node-side filter is any class that implements the interface `IlvNodeSideFilter`. This interface defines the following method:.

```
public boolean accept(IlvGraphModel graphModel,
                      Object link,
                      boolean origin,
                      Object node,
                      int side);
```

This method allows you to permit the input `link` to connect its `origin` or destination to the input `side` of the input `node`.

As an example, assume that the application requires that for end nodes of type `IlvShadowRectangle`, links can connect their origin only at the top and bottom side. For

end nodes of type `IlvReliefRectangle`, links can connect their destination only at the left and right side. The following node-side filter accomplishes this:

```
class MyFilter implements IlvNodeSideFilter
{
     public boolean accept(IlvGraphModel graphModel,
                           Object link,
                           boolean origin,
                           Object node,
                           int side)
    {
        if (node instanceof IlvShadowRectangle && origin)
            return(side == IlvDirection.Top || side == IlvDirection.Bottom);
        if (node instanceof IlvReliefRectangle && !origin)
            return(side == IlvDirection.Left || side == IlvDirection.Right);
        return true;
    }
}
```

To set this node-side filter, use the following call:

```
layout.setNodeSideFilter(new MyFilter());
```

To remove the node-side filter, use the following call:

```
layout.setNodeSideFilter(null);
```

Using a Node Box Interface

Some applications require that effective area of a node is not exactly its bounding box. For instance, if the node has a shadow, the shadow is included in the bounding box. However, the shadow may not be considered as an obstacle for the links. In this case, the effective bounding box of a node is smaller than the bounding box returned by `IlvGraphic.boundingBox`.

You can modify the effective bounding box of a node by implementing a class that implements the `IlvNodeBoxInterface`. This interface defines the following method:

```
public IlvRect getBox(IlvGraphModel graphModel, Object node);
```

This method allows you to return the effective bounding box. For instance, to set a node box interface that returns a smaller bounding box for all nodes of type `IlvShadowRectangle`, call:

```
layout.setNodeBoxInterface(new IlvNodeBoxInterface() {
    public IlvRect getBox(IlvGraphModel graphModel, Object node) {
        IlvRect rect = graphModel.boundingBox(node);
        if (node instanceof IlvShadowRectangle) {
            // need a rect that is 4 units smaller
            rect.resize(rect.width-4.f, rect.height-4.f);
        }
        return rect;
    }
});
```

Using a Link Connection Box Interface

By default, the connection points of the links are distributed on the border of the bounding box of the nodes. Sometimes, it may be necessary to place the connection points on a rectangle that is smaller or larger than the bounding box. For instance, this can happen when labels are displayed below or above nodes.

You can modify the position of the connection points of the links by implementing a class that implements the `IlvLinkConnectionBoxInterface`. This is a subinterface of `IlvNodeBoxInterface` (see *Using a Node Box Interface* on page 207). It defines again the method:

```
public IlvRect getBox(IlvGraphModel graphModel, Object node);
```

This method allows you to return the effective rectangle on which the connection points of the links are placed.

Additionally, the interface `IlvLinkConnectionBoxInterface` defines a second method:

```
public float getTangentialOffset(IlvGraphModel graphModel,
                                 Object node, int nodeSide);
```

This method is used only in the short link mode. For details, see *Using a Link Connection Box Interface* on page 211. When using the Link Layout in long link mode, just implement the method by returning the value 0.

For Experts: Special Options of the Short Link Layout

The Link Layout algorithm utilizes the class `IlvShortLinkLayout` as a subalgorithm. `IlvShortLinkLayout` is a subclass of `IlvGraphLayout` and can be used a stand-alone as well. To access the instance of `IlvShortLinkLayout` that is used by the Link Layout algorithm, call:

```
IlvShortLinkLayout getShortLinkLayout();
```

Using this accessor, you can control many special features of the Short Link Layout that are not made available by the `IlvLinkLayout` class because these features are for experts only.

Self-link Style

Self-links are links with the same origin and destination nodes. The Short Link Layout provides two optional shapes for the self-links.

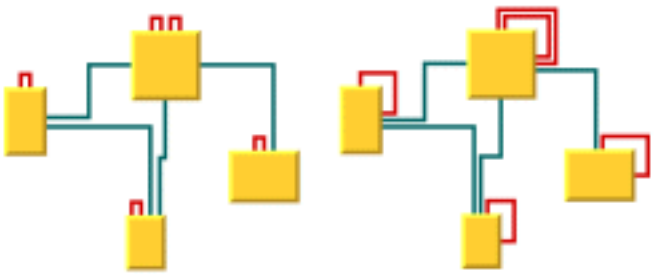


Figure 4.98 *Self-link Style Options*

To set and retrieve the style of the self-links, use the following methods:

```
void setGlobalSelfLinkStyle(int style)
int getGlobalSelfLinkStyle()
```

The valid values for `style` are:

- `IlvShortLinkLayout.TWO_BENDS_ORTHOGONAL_STYLE`
- `IlvShortLinkLayout.THREE_BENDS_ORTHOGONAL_STYLE`

Number of Optimization Iterations

The link shape optimization is stopped if the time exceeds the allowed time (see *Allowed Time* on page 192) or if the number of iterations exceeds the allowed number of iterations. To set this number to 3, call:

```
layout.getShortLinkLayout().setAllowedNumberOfIterations(3);
```

> ***Note:*** *You may want to disable the link shape optimization by setting the number of iterations to zero in order to increase the speed of the layout process.*

Same Shape for Multiple Links

You can force the layout algorithm to compute the same shape for all the links having common origin and destination nodes. The links will have parallel shapes.

When this option is disabled, the layout is free to compute different shapes for links connecting the same pair of nodes. Generally, different shapes are chosen to avoid some overlaps.

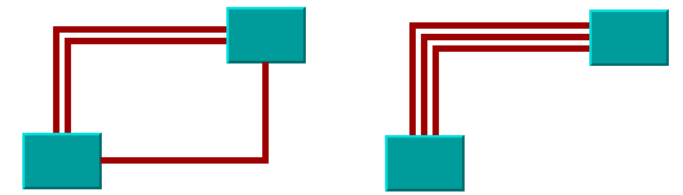


Figure 4.99 *Self-link Style Options*

To enable this option, use the method:

```
layout.getShortLinkLayout().setSameShapeForMultipleLinks(true);
```

The default value is `false`.

Link Crossing Penalty

The computation of the shape of the links is driven by the objective to minimize a cost function, which is proportional to the number of link-to-link crossings and link-to-node crossings. By default, these two types of crossings have equal weights of `1`. You can increase the weight of the **link-to-node crossings** (for instance, to a value of `5`) by calling:

```
layout.getShortLinkLayout().setLinkToNodeCrossingPenalty(5.f);
```

This increases the possibility of obtaining a layout with no link-to-node crossings (or with only a few crossings), with the expense that there may be more link-to-link crossings.

You can alternatively increase the weight of the **link-to-link crossings** (for instance, to a value of `3`) by calling:

```
layout.getShortLinkLayout().setLinkToLinkCrossingPenalty(3.f);
```

This increases the possibility of obtaining a layout with no link-to-link crossings (or with a only few crossings), with the expense that there may be more link-to-node crossings.

Bypass Distance

If the origin and destination nodes are too close, there may not be enough space for routing the link directly between the end nodes. Therefore, by default, if the end nodes are closer than a threshold distance, the layout choses link shapes that bypass the interval between close nodes. (See Figure 4.100.)

End-nodes distance larger than the bypass distance

End-nodes distance smaller than the bypass distance

Figure 4.100 *End Nodes and Bypass Distance*

The bypass distance is the minimum distance between the origin and destination nodes for which a link shape going directly from one node to another is allowed. The algorithm tries to avoid link shapes that connect directly the sides of the end nodes that are closer than the bypass value.

To set and retrieve the bypass distance, use the following methods:

```
void setBypassDistance(float dist)
```

```
float getBypassDistance()
```

The default value is a strictly negative value. If the bypass distance is strictly negative, the value of the **minimum final segment length** (see *Minimum Final Segment Length* on page 206) parameter is used as the bypass distance. This allows the automatic adjustment of the bypass distance according to the current value of the minimum final segment length. This behavior is suitable in most cases. However, you can specify a non-negative value in order to override the default behavior.

Using a Link Connection Box Interface

By default, the connection points of the links are distributed on the border of the bounding box of the nodes, symmetrically with respect to the middle of each side. Sometimes, it may be necessary to place the connection points on a rectangle smaller or larger than the bounding box, eventually in a nonsymmetric way. For instance, this can happen when labels are displayed below or above nodes.

You can modify the position of the connection points of the links by implementing a class that implements the `IlvLinkConnectionBoxInterface`. This interface defines the following method:

```
public IlvRect getBox(IlvGraphModel graphModel, Object node);
```

This method allows you to return the effective rectangle on which the connection points of the links are placed.

A second method defined on the interface allows the connection points to be "shifted" tangentially, in a different way for each side of each node:

```
public float getTangentialOffset(IlvGraphModel graphModel,
                                 Object node, int nodeSide);
```

For instance, to set a link connection box interface that returns a link connection rectangle that is smaller than the bounding box for all nodes of type `IlvShadowRectangle` and shifts up the connection points on the left and right side of all the nodes, call:

```
layout.setLinkConnectionBoxInterface(new IlvLinkConnectionBoxInterface() {
    public IlvRect getBox(IlvGraphModel graphModel, Object node) {
        IlvRect rect = graphModel.boundingBox(node);
        if (node instanceof IlvShadowRectangle) {
            // need a rect that is 4 pixels smaller
            rect.resize(rect.width-4.f, rect.height-4.f);
        }
        return rect;
    }
    public float getTangentialOffset(IlvGraphModel graphModel,
                                     Object node, int nodeSide) {
        switch (nodeSide) {
          case IlvDirection.Left:
          case IlvDirection.Right:
            return -10; // shift up with 10 for both left and right side
          case IlvDirection.Top:
          case IlvDirection.Bottom:
          default:
            return 0; // no shift for top and bottom side
        }
    }
});
```

Figure 4.98 shows the effects of customizing the connection box. On the left is the result using the default settings: the connection points are distributed on the bounding box of the node (which includes the label) and are symmetric with the middle of each node side (including the label). On the right, is the result after specifying a link connection box interface. On the bottom side of the nodes, the links are now connected to the node (passing over the label), while on the left and right side the nodes are now symmetric to the middle of the node (without the label).

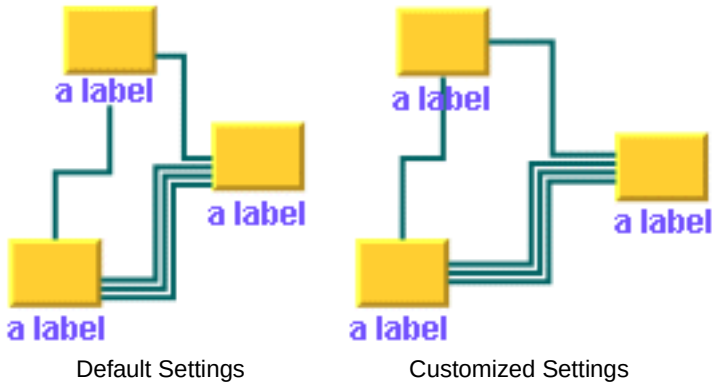


Figure 4.101 *Customization of the Link Connection Box*

For Experts: Special Options of the Long Link Layout

The Link Layout algorithm utilizes the class `IlvLongLinkLayout` as subalgorithm. `IlvLongLinkLayout` is a subclass of `IlvGraphLayout` and can be used a stand-alone as well. To access the instance of `IlvLongLinkLayout` that is used by the Link Layout algorithm, use the method:

```
IlvLongLinkLayout getLongLinkLayout();
```

Using this accessor, you can control many special features of the Long Link Layout that are not made available by the `IlvLinkLayout` class because these features are for experts only.

Specifying Additional Obstacles

The Long Link Layout algorithm considers nodes to be obstacles that cannot be overlapped and links to be obstacles that can be crossed at an angle of 90 degree (approximately, if the link style is direct), but that cannot be overlapped.

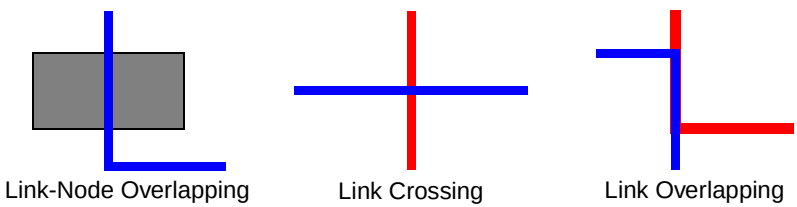


Figure 4.102 *Crossings and Overlappings*

If an application requires additional obstacles that are not links or nodes, these can be specified by the following:

```
layout.getLongLinkLayout().addRectObstacle(rect);

layout.getLongLinkLayout().addLineObstacle(point1, point2);
```

Rectangular obstacles behave like nodes: links cannot overlap the rectangles. Line obstacles behave like link segments: other links can cross the line segments, but cannot overlap the segments. These obstacle settings can be removed by the following:

```
layout.getLongLinkLayout().removeAllRectObstacles();

layout.getLongLinkLayout().removeAllLineObstacles();
```

Penalties for Variable End Points

If the termination points of the links are not fixed, the algorithm uses a heuristic to calculate the termination points of each link. It examines all free grid points that are close to the border of the start and end node and assigns a penalty to each grid point. If a node-side filter is installed, the penalty depends on whether the node side is allowed or rejected.

A more precise way to affect how the termination points are chosen is the termination point filter. This enables the user to specify the penalty for each grid point. A termination point filter is a class that implements the interface `IlvTerminationPointFilter` that defines the following method:

```
public int getPenalty(IlvGraphModel graphModel, Object link,
                      boolean origin, Object node, IlvPoint point,
                      int side, int proposedPenalty);
```

To select the `origin` or destination point of the input `link`, the input `point` (a grid point on the input `side` of the `node`) is examined. The `proposedPenalty` is calculated by the default heuristic of the algorithm. You can return a changed penalty or you can return `java.lang.Integer.MAX_VALUE` in order to reject the grid point. If the grid point is rejected, it is not chosen as termination point of the link.

The termination point filter can be set by the following:

```
layout.getLongLinkLayout().setTerminationPointFilter(MyFilter);
```

Manipulating the Routing Phases

As mentioned in *Long Link Layout Algorithm* on page 190, the algorithm first treats each link individually and the applies a crossing reduction phase to all links. To find a route for an individual link, the algorithm first checks whether a routing (such as a straight line or with only one bend) is possible. If this kind of routing is not possible, it uses a sophisticated, but more time consuming, grid search algorithm with backtracking to find a route with many bends.

To switch off the phase that finds a straight-line or one-bend routing, use the following:

```
layout.getLongLinkLayout().setStraightRouteEnabled(false);
```

The backtrack search for a route with many bends can be affected in the several ways.

- You can specify the maximal number of backtrack steps by using the following:

```
layout.getLongLinkLayout().setMaxBacktrack(1000);
```

The default maximal backtrack number is 30000,

- A more convenient way is to specify the maximal time available to search for the route for each link:

  ```
  layout.getLongLinkLayout().setAllowedTimePerLink(4000);
  ```

  The default allowed time per link is 2000 milliseconds (2 seconds).

- Finally, you can specify how many steps should be done during the crossing reduction phase by using:

  ```
  layout.getLongLinkLayout().setNumberCrossingReductionIterations(5);
  ```

  You can disable the crossing reduction completely by using the following:

  ```
  layout.getLongLinkLayout().setCrossingReductionEnabled(false);
  ```

Fallback Mechanism

The Long Link Layout algorithm may not be able to find a routing for a link, if one end nodes is inside an enclave. In Figure 4.103, the red node is inside an enclave. In this case, the backtrack search algorithm fails to find a routing without overlapping nodes. The backtrack search algorithm may also fail if the situation is so complex that the search exceeds the allowed time per link.

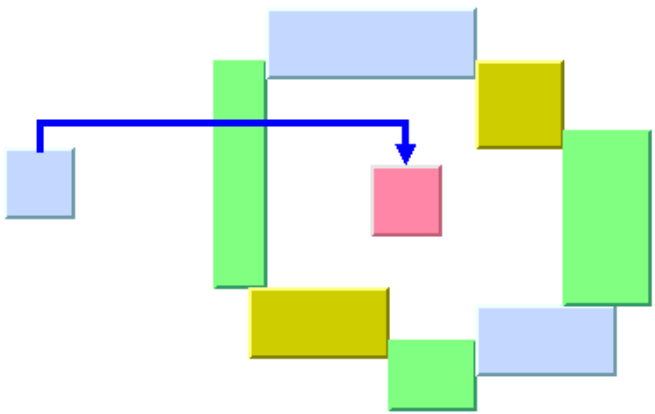


Figure 4.103 *A Node Inside an Enclave*

When the backtrack search algorithm fails to find a routing, a simple fallback mechanism is applied that creates a routing with node overlappings. This fallback mechanism can be disabled by using the following:

```
layout.getLongLinkLayout().setFallbackRouteEnabled(false);
```

If the fallback mechanism is disabled, these links are not routed at all and remain in the same shape as before the layout.

To retrieve the links that could not be routed in the usual way without the fallback mechanism, you can use the following:

```
Enumeration e = layout.getLongLinkLayout().getCalcFallbackLinks();
```

For instance, you can iterate over these links and apply your own specific fallback mechanism instead of the default fallback mechanism of the Long Link Layout algorithm.

Random Layout

In this section, you will learn about the *Random Layout* algorithm from the ILOG JViews Graph Layout module (class `IlvRandomLayout` from the package `ilog.views.graphlayout.random`).

Sample

Here is a sample drawing produced with the Random Layout:

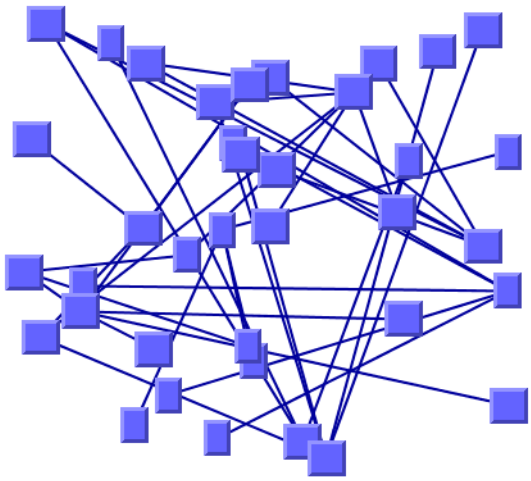


Figure 4.104 *Graph Drawing Produced with the Random Layout*

What Types of Graphs?

Any type of graph:

- Connected and disconnected graphs
- Planar and nonplanar graphs

Features

Random placement of the nodes of a grapher inside a given region.

Limitations

- The algorithm computes random coordinates for the upper-left corner of the graphic objects representing the nodes. In some cases, this may not be appropriate.
- To ensure that the nodes do not overlap the margins of the layout region, the algorithm computes the coordinates randomly inside a region whose width and height are smaller than the width and height of the layout region. The difference is the maximum width and the maximum height of the nodes, respectively. In some cases, this may not be appropriate.

Brief Description of the Algorithm

The Random Layout algorithm is not really a layout algorithm. It simply places the nodes at randomly computed positions inside a user-defined region. Nevertheless, the Random Layout algorithm may be useful when a random, initial placement is needed by another layout algorithm or in cases where an aesthetic, readable drawing is not important.

Code Sample

Below is a code sample using the `IlvRandomLayout` class:

```
 ...
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.random.*;
 ...
IlvGrapher grapher = new IlvGrapher();
IlvManagerView view = new IlvManagerView(grapher);

 ... /*  Fill in here the grapher with nodes and links in */

IlvRandomLayout layout = new IlvRandomLayout();
layout.attach(grapher);
try {
        IlvGraphLayoutReport layoutReport = layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Generic Features and Parameters

The `IlvRandomLayout` class supports the following generic parameters defined in the `IlvGraphLayout` class:

- *Layout Region*
- *Percentage Completion Calculation*
- *Preserve Fixed Links*
- *Preserve Fixed Nodes*
- *Random Generator Seed Value*
- *Save Parameters to Named Properties*
- *Stop Immediately*

The following comments describe the particular way in which these parameters are used by this subclass.

Layout Region

The layout algorithm uses the layout region setting (either your own or the default setting) to control the size and the position of the graph drawing. All three ways to specify the layout region are available for this subclass. (See *Layout Region* on page 25.)

Percentage Completion Calculation

The layout algorithm calculates the estimated percentage of completion. This value can be obtained from the layout report during the run of the layout. (For a detailed description of this features, see *Percentage Completion Calculation* on page 31 and *Layout Event Listeners* on page 294.)

Preserve Fixed Links

The layout algorithm does not reshape the links that are specified as fixed. (See *Preserve Fixed Links* on page 32.)

Preserve Fixed Nodes

The layout algorithm does not move the nodes that are specified as fixed. (See *Preserve Fixed Nodes* on page 32.)

Random Generator Seed Value

The Random Layout uses a random number generator to compute the coordinates. You can specify a particular value to be used as a seed value. (See *Random Generator Seed Value* on page 33.) For the default behavior, the random generator is initialized using the current system clock. Therefore, different layouts are obtained if you perform the layout repeatedly on the same graph.

Save Parameters to Named Properties

The layout algorithm can save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Save Parameters to Named Properties* on page 34 and *Saving Layout Parameters and Preferred Layouts* on page 315.)

Stop Immediately

The layout algorithm stops after cleanup if the method `stopImmediately` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop Immediately* on page 35.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific Parameters

The following parameters are specific to the `IlvRandomLayout` class:

Link Style

When the layout algorithm moves the nodes, straight-line links (such as instances of `IlvLinkImage`) will automatically "follow" the new positions of their end nodes. If the grapher contains other types of links (for example, `IlvPolylineLinkImage` or `IlvSplineLinkImage`), the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can ask the layout algorithm to automatically remove all the intermediate points of the links (if any). To do this, the following method is provided:

```
void setLinkStyle(int style)
```

The valid values for `style` are:

- `IlvRandomLayout.NO_RESHAPE_STYLE`

  None of the links is reshaped in any manner.

- `IlvRandomLayout.STRAIGHT_LINE_STYLE`

  All the intermediate points of the links (if any) are removed. This is the default value.

> ***Note:*** *If the* `STRAIGHT_LINE_STYLE` *is selected, the layout algorithm may raise an* `IlvInappropriateLinkException` *if layout is performed on an* `IlvGrapher`*, but links that cannot be given a straight-line shape are used. Such links are* `IlvOneLinkImage`*,* `IlvDoubleLinkImage`*,* `IlvOneSplineLinkImage`*, and* `IlvDoubleSplineLinkImage`*. You can use the method* `EnsureAppropriateLinkTypes` *or* `EnsureAppropriateLinks` *defined in* `IlvGraphLayoutUtil` *to automatically replace inappropriate links. This can be done before layout or when the* `IlvInappropriateLinkException` *is caught. For details on these methods, see the ILOG JViews Graph Layout Reference Manual. For details on the graph model, see Using the Graph Model on page 297.*

To obtain the current choice, use the following method:

```
int getLinkStyle()
```

Bus Layout

In this section, you will learn about the *Bus Layout* algorithm from the ILOG JViews Graph Layout module (class `IlvBusLayout` from the package `ilog.views.graphlayout.bus`).

Sample

Here is a sample drawing produced with the Bus Layout:

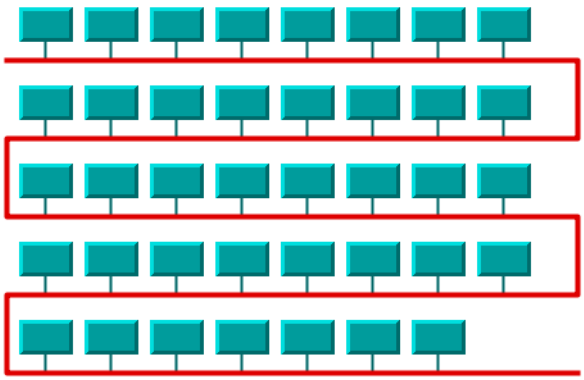


Figure 4.105 *Bus Topology Produced with the Bus Layout*

What Types of Graphs?

- Bus network topologies (a set of nodes connected to a bus object)

Application Domains

Application domains of the Bus Layout include:

- Telecom and networking (LAN diagrams)
- Electrical engineering (circuit block diagrams)
- Industrial engineering (equipment/resource control charts)

Features

- Displays bus topologies.
- Takes into account the size of the nodes so that no overlapping occurs.
- Provides several ordering, alignment, and flow direction options.
- Allows easy customization of the dimensional parameters.

Brief Description of the Algorithm

Bus topology is well known in network management and telecommunications fields. The Bus Layout class can display these topologies nicely. It represents the "bus" as a "serpent" polyline. The width of the "serpent" is user-defined (via the width of the layout region parameter) and the height is computed so that enough space is available for all the nodes.

Code Sample

Below is a code sample using the `IlvBusLayout` class:

```
 ...
import ilog.views.*;
import ilog.views.graphic.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.bus.*;
 ...
IlvGrapher grapher = new IlvGrapher();
IlvManagerView view = new IlvManagerView(grapher);

 ... /*  Fill in the grapher with nodes and links here */

/* Create the bus node; the number of points and
   the coordinates are not important */
IlvPoint[] points = {new IlvPoint(10, 10)};
IlvPolyline bus = new IlvPolyline(points);
grapher.addNode(bus, false);

 ... /* Fill in the grapher with links between each node
       and the bus here */

IlvBusLayout layout = new IlvBusLayout();
layout.attach(grapher);

/* Specify the bus node */
layout.setBus(bus);
```

```
try {
        IlvGraphLayoutReport layoutReport = layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Generic Features and Parameters

The `IlvBusLayout` class supports the following generic parameters defined in the `IlvGraphLayout` class:

- *Allowed Time*
- *Layout of Connected Components*
- *Layout Region*
- *Link Clipping*
- *Preserve Fixed Links*
- *Preserve Fixed Nodes*
- *Save Parameters to Named Properties*
- *Stop Immediately*

The following comments describe the particular way in which these parameters are used by this subclass.

Allowed Time

The layout algorithm stops if the allowed time setting has elapsed. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Allowed Time* on page 22.) The result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Layout of Connected Components

The layout algorithm can utilize the generic mechanism to layout connected components. (For more information about this mechanism, see *Layout of Connected Components* on page 25.)

Layout Region

The layout algorithm uses the layout region setting (either your own or the default setting) to control the size and the position of the graph drawing. All three ways to specify the layout region are available for this subclass. (See *Layout Region* on page 25.)

The size of the layout is chosen with respect to the layout region (see Figure 4.113 on page 234). The height of the layout region is not taken into account. The height of the layout will be smaller or larger, depending on the number of nodes, the size of the nodes, and the other specified parameters.

Link Clipping

The layout algorithm can use a link clip interface to clip the end points of a link. (See *Link Clipping* on page 27.)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip interface is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using a Link Clipping Interface* on page 236 for details of the link clipping mechanism.

Preserve Fixed Links

The layout algorithm does not reshape the links that are specified as fixed. (See *Preserve Fixed Links* on page 32.)

Preserve Fixed Nodes

The layout algorithm does not move the nodes that are specified as fixed. (See *Preserve Fixed Nodes* on page 32.)

Save Parameters to Named Properties

The layout algorithm can save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Save Parameters to Named Properties* on page 34 and *Saving Layout Parameters and Preferred Layouts* on page 315.)

Stop Immediately

The layout algorithm stops after cleanup if the method `stopImmediately` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop Immediately* on page 35.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific Parameters

The following parameters are specific to the `IlvBusLayout` class.

Order Parameter

The order parameter specifies how to arrange the nodes.

To specify the ordering option for the nodes, use the method:

```
void setOrdering(int ordering)
```

The valid values for `ordering` are:

- `IlvBusLayout.NO_ORDERING`

  The nodes are arranged on the bus in an arbitrary order. This is the default value.

- `IlvBusLayout.ORDER_BY_HEIGHT`

  The nodes are arranged on the bus according to height, starting at the upper-left corner of the bus with the tallest node. This option can save vertical space.

- `IlvBusLayout.ORDER_BY_INDEX`

  This option allows you to specify a particular order for the nodes. The nodes are arranged on the bus according to their index values, starting at the upper-left corner of the bus with the node with the smallest index. Nodes for which you do not specify an index are placed after the nodes for which an index is specified.

  The `IlvBusLayout.ORDER_BY_INDEX` option allows you to specify the order of the nodes, according to physical, geographical data for example. If this option is chosen, the algorithm sorts the nodes in ascending order according to their index values.

To obtain the current value, use the method:

```
int getOrdering()
```

The index is an integer value associated with a node and is specified using the method:

```
void setIndex(Object node, int index)
```

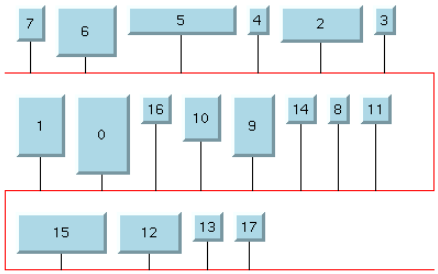
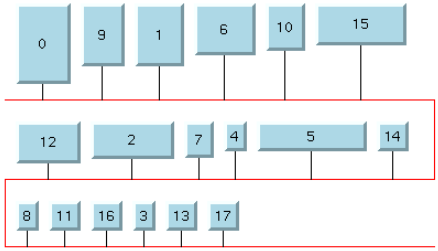
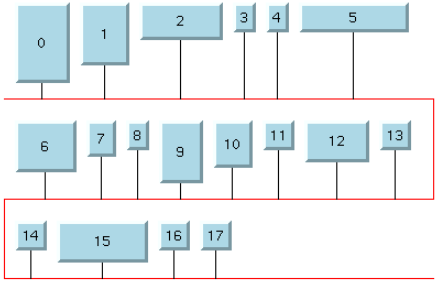
The values of the indices cannot be negative. To obtain the current index of a node, use the method:

```
int getIndex(Object node)
```

If no index is specified for the node, the value `IlvBusLayout.NO_INDEX` is returned.

The ordering options for the Bus Layout are illustrated in Table 4.4.

Table 4.4 *Ordering Options the Nodes for the Bus Layout Algorithm*

Ordering	Layout
NO_ORDERING	
ORDER_BY_HEIGHT	
ORDER_BY_INDEX	

Bus Node

To represent bus topologies, the algorithm reshapes a special node, called the "bus node", and gives it a "serpent" form. This bus node must be an instance of the `IlvPolyPointsInterface` class. Usually, you use its subclass `IlvPolyline`. Before performing the layout, you must create this object and add it to the grapher as a node. (The number of points in the object you create is not important.) Then, you must specify the node as "bus node" using the methods:

```
void setBus(IlvPolyPointsInterface bus)
```

```
IlvPolyPointsInterface getBus()
```

If none is specified, the Bus layout automatically tries to find an appropriate node that can be used as bus object.

The bus object must implement the interface `IlvPolyPointsInterface` and it must allow the insertion and removal of points (see the methods `allowsPointInsertion` and `allowsPointRemoval` defined by the interface). The initial number of points is not significant. When a bus object is specified or automatically discovered in an `IlvGrapher`, the appropriate link connector is automatically installed on it. By default, the link connector is of type `IlvBusLinkConnector`.

Usually, the class `IlvPolyline` is used for the bus object. The bus object must be added to the `IlvGrapher` as a node (using the method `addNode`). The links between the bus and the nodes connected to the bus must be created before performing the layout. (See the code provided in *Code Sample* on page 223.)

Link Style

When the layout algorithm moves the nodes, straight-line links (such as instances of `IlvLinkImage`) will automatically "follow" the new positions of their end nodes. If the grapher contains other types of links (for example, `IlvPolylineLinkImage` or `IlvSplineLinkImage`), the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can ask the layout algorithm to automatically remove all the intermediate points of the links (if any). To do this, the following method is provided:

```
void setLinkStyle(int style)
```

The valid values for `style` are:

- `IlvBusLayout.NO_RESHAPE_STYLE`

  None of the links is reshaped in any manner.

- `IlvBusLayout.STRAIGHT_LINE_STYLE`

  All the intermediate points of the links (if any) are removed. This is the default value.

Note: *If the* `STRAIGHT_LINE_STYLE` *is selected, the layout algorithm may raise an* `IlvInappropriateLinkException` *if layout is performed on an* `IlvGrapher`*, but links that cannot be given a straight-line shape are used. Such links are* `IlvOneLinkImage`*,* `IlvDoubleLinkImage`*,* `IlvOneSplineLinkImage`*, and* `IlvDoubleSplineLinkImage`*. You can use the methods* `EnsureAppropriateLinkTypes` *or* `EnsureAppropriateLinks` *defined in* `IlvGraphLayoutUtil` *to automatically replace inappropriate links. This can be done before layout or when the* `IlvInappropriateLinkException` *is caught. For details on these methods, see the ILOG JViews Graph Layout Reference Manual. For details on the graph model, see Using the Graph Model on page 297.*

To obtain the current choice, use the following method:

```
long getLinkStyle()
```

Flow Direction

The flow direction options control the horizontal alignment of each row (bus level) with respect to the left and right sides of the layout region. The rows can be either all left-aligned on the left border of the layout region or can alternate between the left and right alignment.

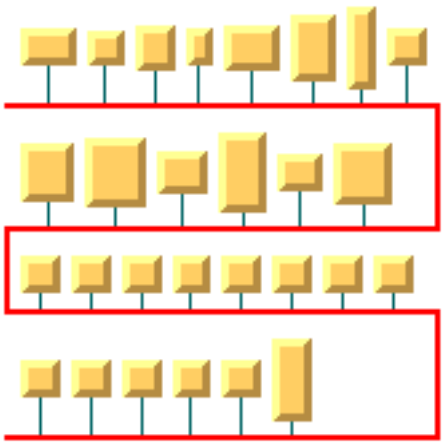


Figure 4.106 *Bus Layout with Left-to-Right Flow Direction*

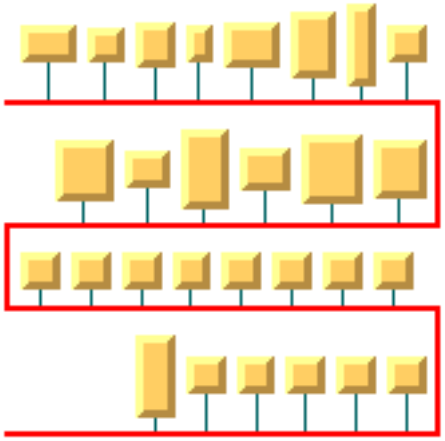


Figure 4.107 *Bus Layout with Alternate Flow Direction*

To set and retrieve the flow direction, use the methods:

```
void setFlowDirection(int direction);

int getFlowDirection();
```

The valid values for `direction` are:

- `IlvBusLayout.LEFT_TO_RIGHT` (the default)

  All the rows (bus levels) are left-aligned.

- `IlvBusLayout.ALTERNATE`

  The even rows (bus levels) are left-aligned and the odd rows are right-aligned.

Maximum Number of Nodes Per Level

By default, the layout places as many nodes on each level as possible given the size of the nodes and the dimensional parameters (layout region and margins). If needed, the layout can additionally respect a specified maximum number of nodes per level (see Figure 4.108 on page 231 and Figure 4.109 on page 231).

To set and retrieve the maximum number of nodes per level, use the methods:

```
void setMaxNumberOfNodesPerLevel(int nNodes);

int getMaxNumberOfNodesPerLevel();
```

The default value is `Integer.MAX_VALUE`, that is the number of nodes placed in each level is only bounded by the size of the nodes and the dimensional parameters. The specified value must be at least `1`.

Bus Width Adjusting

By default, the width of the bus object, that is the difference between the maximum and minimum x-coordinates, depends on the width of the layout region and the other dimensional parameters (see Figure 4.113, *Bus Layout with Bottom Vertical Alignment* on page 234). Optionally, the width of the bus object can be automatically adjusted to the total width of the nodes, plus the offsets and the margins. This option can be particularly useful in conjunction with the customization of the maximum number of nodes per level (see *Maximum Number of Nodes Per Level* on page 230).

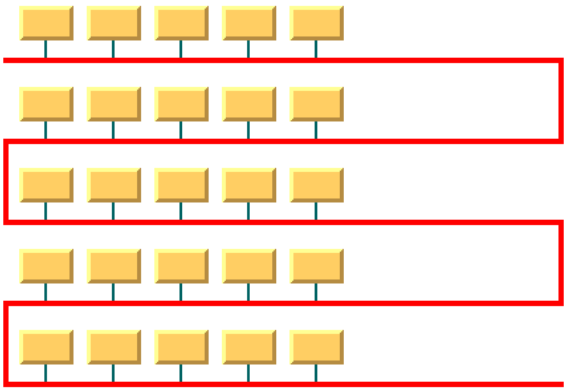


Figure 4.108 *Bus Width Adjusting Disabled and Bounded Number of Nodes per Level*

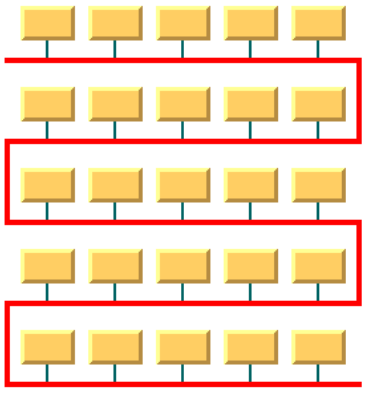


Figure 4.109 *Bus Width Adjusting Enabled and Bounded Number of Nodes per Level*

To enable or disable the bus width adjusting, use the method:

```
void setBusWidthAdjustingEnabledM(boolean enable);
```

To get the current value, use the method:

```
boolean isBusWidthAdjustingEnabled();
```

The bus width adjusting is disabled by default.

Alignment Parameters

The alignment options control how a node is placed above its row (bus level). The alignment can be set globally, in which case all nodes are aligned in the same way, or locally on each node, with the result that different alignments occur in the same drawing.

Global Alignment Parameters

To set and retrieve the global alignment, use the following methods:

```
void setGlobalVerticalAlignment(int alignment);
```

```
int getGlobalVerticalAlignment();
```

The valid values for `alignment` are:

- `IlvBusLayout.CENTER` (the default)

  The node is vertically centered over its row (bus level).

- `IlvBusLayout.TOP`

  The node is vertically aligned on the top of its row (bus level).

- `IlvBusLayout.BOTTOM`

  The node is vertically aligned on the bottom of its row (bus level).

- `IlvBusLayout.MIXED`

  Each node can have a different alignment. The alignment of each individual node can be set with the result that different alignments can occur in the same graph.

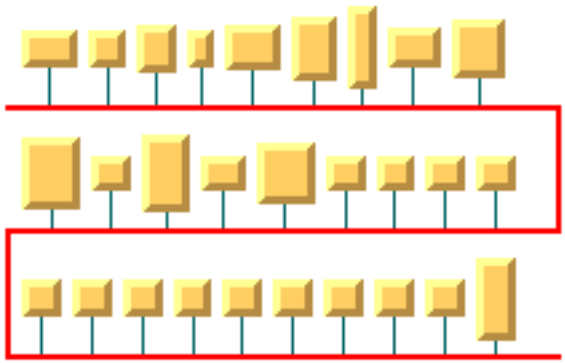


Figure 4.110 *Bus Layout with Center Vertical Alignment*

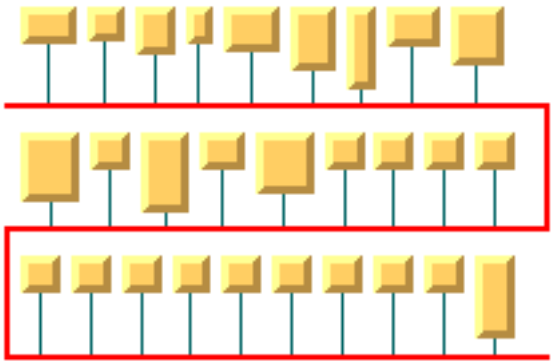


Figure 4.111 *Bus Layout with Top Vertical Alignment*

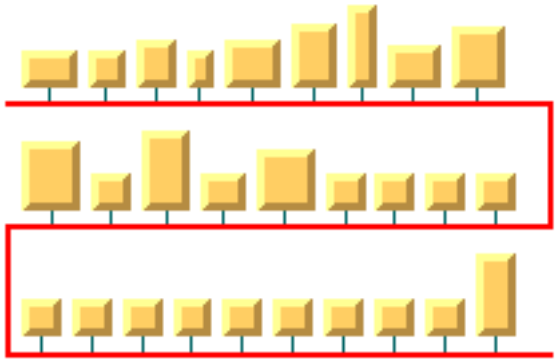


Figure 4.112 *Bus Layout with Bottom Vertical Alignment*

Alignment of Individual Nodes

All nodes have the same alignment unless the global alignment is set to `IlvBusLayout.MIXED`. Only when the global alignment is set to mixed can each node have an individual alignment style.

To set and retrieve the alignment of an individual node, use the following methods:

```
void setVerticalAlignment(Object node, int alignment);

int getVerticalAlignment(Object node);
```

The valid values for `alignment` of a node are:

- `IlvBusLayout.CENTER` (the default)
- `IlvBusLayout.TOP`

- `IlvBusLayout.BOTTOM`

Incremental Mode

The Bus Layout algorithm normally places all the nodes from scratch. If the graph incrementally changes because you add, remove, or resize nodes, the subsequent layout may differ considerably from the previous layout. To avoid this effect and to help the user to retain a mental map of the graph, the algorithm has an incremental mode.In incremental mode, the layout tries to place the nodes at the same location or in the same order as in the previous layout whenever it is possible

To enable the incremental mode, call:

```
layout.setIncrementalMode(true);
```

> ***Note:*** *To preserve the stability, the incremental mode can keep some regions free. Therefore, the total area of the layout can be larger than in nonincremental mode, and, in general, the layout may not look as nice as in nonincremental mode.*

Dimensional Parameters

Figure 4.113 illustrates the dimensional parameters used in the Bus Layout algorithm. These parameters are explained in the sections that follow.

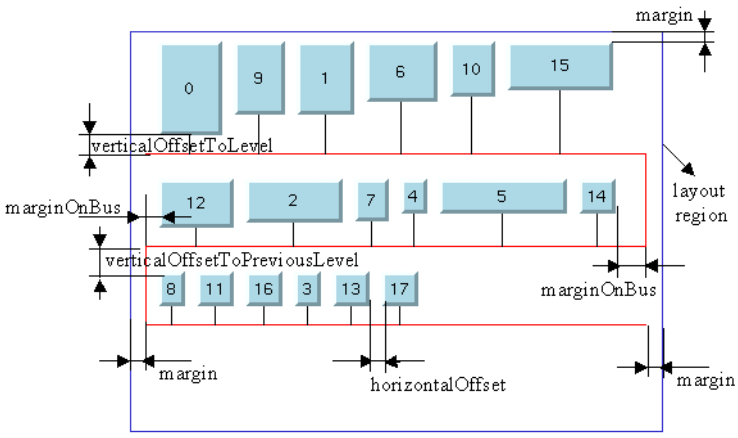


Figure 4.113 *Dimensional Parameters for the Bus Layout Algorithm*

Horizontal Offset

This parameter represents the horizontal distance between two nodes. To specify the horizontal offset, use the method:

```
void setHorizontalOffset(float offset)
```

To obtain the current value, use the method:

```
float getHorizontalOffset()
```

Vertical Offset to Level

This parameter represents the vertical distance between a row of nodes and the next horizontal segment of the bus node. To specify this parameter, use the method:

```
void setVerticalOffsetToLevel(float offset)
```

To obtain the current value, use the method:

```
float getVerticalOffsetToLevel()
```

Vertical Offset to Previous Level

This parameter represents the vertical distance between a row of nodes and the previous horizontal segment of the bus node. To specify this parameter, use the method:

```
void setVerticalOffsetToPreviousLevel(float offset)
```

To obtain the current value, use the method:

```
float getVerticalOffsetToPreviousLevel()
```

Margin

This parameter represents the offset distance between the layout region and the bounding rectangle of the layout. To specify this parameter, use the method:

```
void setMargin(float margin)
```

To obtain the current value, use the method:

```
float getMargin()
```

Margin on Bus

On the odd horizontal levels (first, third, fifth, and so on) of the bus starting from the top, this parameter represents the offset distance between the left side of the first node on the left and the left side of the bus object.

On the even horizontal levels (second, fourth, sixth, and so on) of the bus starting from the top, this parameter represents the offset distance between the right side of the last node on the right and the right side of the bus object. (See Figure 4.113 on page 234 for an illustration of the margin-on-bus parameter.)

To specify this parameter, use the method:

```
void setMarginOnBus(float margin)
```

To obtain the current value, use the method:

```
float getMarginOnBus()
```

Using a Link Clipping Interface

By default, the Bus Layout does not place the connection points of links at the nodes. At the bus node, it installs a bus link connector that is responsible for the connection points. At the other nodes, it relies on their link connectors to determine the connection points. If no link connectors are installed at these nodes, the default behavior is to connect to a point at the border of the bounding box of the nodes. If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape. An example is shown in Figure 4.114.

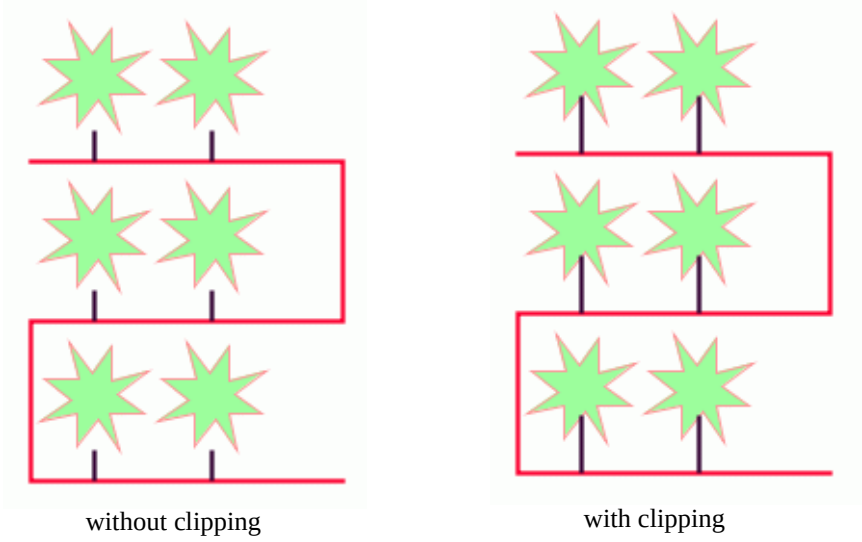


Figure 4.114 *Effect of Link Clipping Interface*

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkClipInterface`. An example for the implementation of a link clip interface is in *Link Clipping* on page 27. To set a link clip interface, call:

```
void setLinkClipInterface(IlvLinkClipInterface interface)
```

Note: *The link clip interface requires link connectors at the nodes of an* `IlvGrapher` *that allow connector pins to be placed freely at the node border. It is recommended that you use* `IlvRelativeLinkConnector` *or* `IlvClipLinkConnector` *for link connectors to be used in combination with* `IlvGrapher` *objects. The clip link connector updates the clipped connection points automatically during interactive node movements.*

The special bus node is an exception: it always uses the bus link connector.

Circular Layout

In this section, you will learn about the *Circular Layout* algorithm from the ILOG JViews Graph Layout module (class `IlvCircularLayout` from the package `ilog.views.graphlayout.circular`).

Samples

Here are sample drawings produced with the Circular Layout:

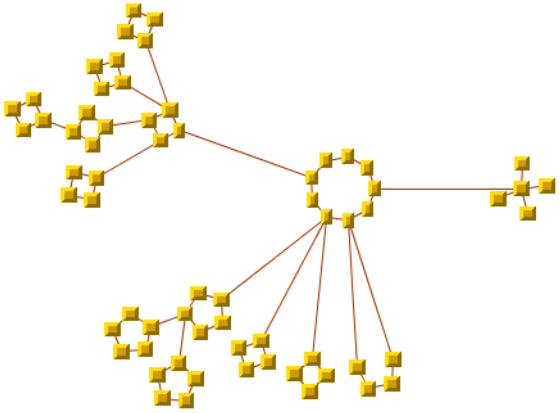


Figure 4.115 *Ring-and-Star Topology Drawing Produced with the Circular Layout*

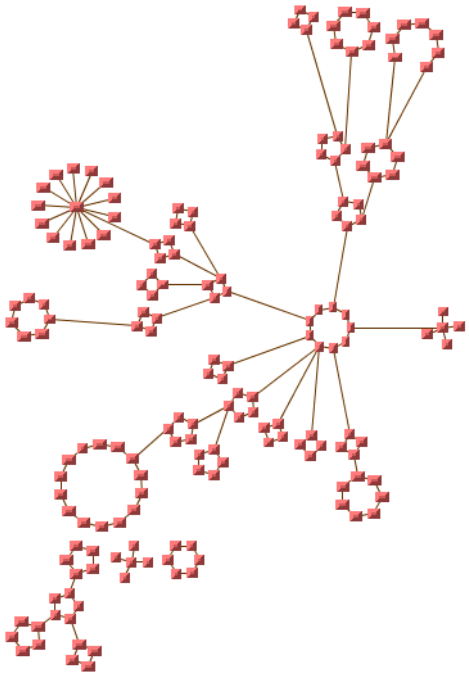


Figure 4.116 *Large Ring-and-Star Topology Drawing Produced with the Circular Layout*

What Types of Graphs?

- Graphs representing interconnected ring and/or star network topologies

Application Domains

Application domains of the Circular Layout include:

- Telecom and networking (LAN diagrams)
- Business processing (organization charts)
- Database and knowledge engineering (sociology, genealogy)
- The World Wide Web (Web hyperlink neighborhood)

Features

- Displays network topologies composed of interconnected rings and/or stars.
- Provides two clustering modes (see *Clustering Mode* on page 245). The first mode lays out clusters as circles and places the clusters. This mode is designed for rings/stars that are interconnected in a tree structure, but it can produce acceptable results even if the graph contains cycles. The second mode lays out the clusters as circles of nodes, minimizing the link crossings while keeping the clusters at their initial position. (See also *Brief Description of the Algorithm* on page 240).
- Takes into account the size of the nodes so that no overlapping occurs.

Limitations

- Link crossings cannot always be avoided.

Brief Description of the Algorithm

Ring and star topologies are similar in several ways. Take a look at Figure 4.117 and Figure 4.118 to get an idea of their similarities.

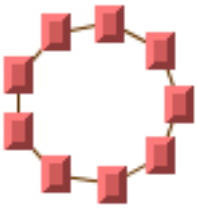


Figure 4.117 *Ring Topology*

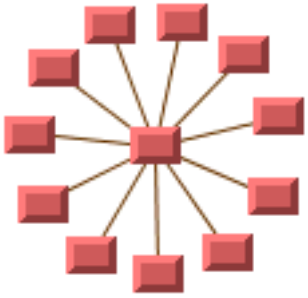


Figure 4.118 *Star Topology*

Both topologies are composed of nodes drawn on a circle. For the Circular Layout algorithm, the only difference between the ring and star topologies is that the star has a special node, called the star center, that is drawn in the center of the circle. The user must specify the node that is the star center. (See *Star Center* on page 247 for information on how to specify the node.)

For each ring or star (generically called a cluster), the Circular Layout algorithm, in one of its modes (see *Clustering Mode* on page 245), allows you to specify the order of the nodes on the circle (this is discussed in *Cluster Membership and Order of the Nodes on a Cluster* on page 245). Otherwise, an arbitrary order is automatically chosen. In another mode, the order is computed automatically such that the number of link crossings is small.

The network topology can be composed of more than one ring or star. These rings and stars can be partially interconnected; that is, two or more clusters can have a common node as shown in Figure 4.119. They can also be interconnected by links between nodes of two different clusters as shown in Figure 4.120.

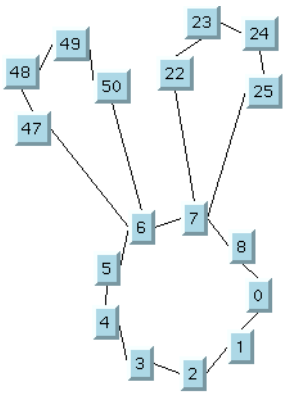


Figure 4.119 *Rings Interconnected by Common Nodes*

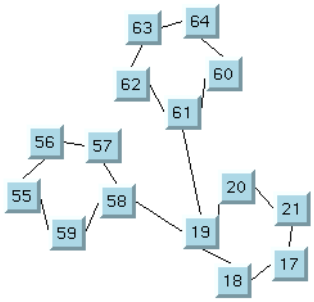


Figure 4.120 *Rings Interconnected by Links*

The Circular Layout algorithm lays out the ring/star topologies in a way that preserves the visual identity of each cluster and avoids overlapping nodes and clusters. (See the sample drawings in *Samples* on page 238.)

To understand how the layout is performed in the clustering mode `BY_CLUSTER_IDS`, consider a graph in which each node represents a ring or star cluster of a network topology. Add a link between two nodes each time there is an interconnection between the corresponding clusters. The Circular Layout algorithm is designed for the case where the graph obtained in this manner is a tree (that is, a graph with no cycles). If cycles exist, the layout is performed using a spanning tree of the graph.

Starting from a root cluster (either a ring or a star), the clusters that are connected to the root cluster are drawn on a circle that is concentric to the root cluster. The radius of the circle is computed to avoid overlapping clusters. Next, the algorithm lays out the clusters connected to these last clusters on a larger circle, and so on. Each circle is called a level.

For networks that are not connected (that is, disconnected groups of clusters exist in the graph), more than one spanning tree exists. Each spanning tree is laid out separately and placed near the others. You can see this in the sample drawings in *Samples* on page 238.

In the clustering mode `BY_SUBGRAPHS`, each subgraph (cluster) keeps its initial position. The subgraphs can be placed either by a different layout algorithm or interactively.

Code Sample

Below is a code sample using the `IlvCircularLayout` class:

```
 ...
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.circular.*;
 ...
IlvGrapher grapher = new IlvGrapher();
IlvManagerView view = new IlvManagerView(grapher);
```

```
IlvCircularLayout layout = new IlvCircularLayout();
layout.attach(grapher);

 ... /* Fill in the grapher with nodes and links here */

// create identifier for cluster 0
IlvClusterNumber clusterId = new IlvClusterNumber(0);

// specify the cluster identifier for cluster 0
// Assume there are three nodes: node1, node2, node3
// the ordering of the nodes: node1 -> node2 -> node3
layout.setClusterId(node1, clusterId, 0); // index 0
layout.setClusterId(node2, clusterId, 1); // index 1
layout.setClusterId(node3, clusterId, 2); // index 2

// create identifier for cluster 1
clusterId = new IlvClusterNumber(1);

// specify the cluster identifier for cluster 1
// Assume there are three nodes: node4, node5, node6
// the ordering of the nodes: node4 -> node5 -> node6
layout.setClusterId(node4, clusterId, 1); // index 1
layout.setClusterId(node5, clusterId, 2); // index 2
layout.setClusterId(node6, clusterId, 0); // index 0

try {
        IlvGraphLayoutReport layoutReport = layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Generic Features and Parameters

The `IlvCircularLayout` class supports the following generic parameters as defined in the `IlvGraphLayout` class:

- *Layout of Connected Components*
- *Layout Region*
- *Link Clipping*
- *Link Connection Box*
- *Preserve Fixed Links*
- *Preserve Fixed Nodes*

- *Save Parameters to Named Properties*
- *Stop Immediately*

The following comments describe the particular way in which these parameters are used by this subclass.

Layout of Connected Components

The layout algorithm can utilize the generic mechanism to layout connected components. (For more information about this mechanism, see *Layout of Connected Components* on page 25).

Layout Region

This parameter has no effect if the clustering mode is `BY_SUBGRAPHS`.

It is not possible to allow the user to control the size of the layout by specifying a bounding box for the drawing. The layout algorithm chooses the size in order to have enough space to avoid overlapping nodes and clusters.

The layout region setting (either your own or the default setting) is used only to choose the position of the center of the drawing. This means that only the center of the layout region is taken into consideration. All three ways to specify the layout region are available for this subclass. (See *Layout Region* on page 25.)

Link Clipping

The layout algorithm can use a link clip interface to clip the end points of a link. (See *Link Clipping* on page 27.)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip interface is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using a Link Clipping Interface* on page 252 for details of the link clipping mechanism.

Link Connection Box

The layout algorithm can use a link connection box interface (see *Link Connection Box* on page 29) in combination with the link clip interface. If no link clip interface is used, the link connection box interface has no effect. For details see *Using a Link Clipping Interface* on page 252.

Preserve Fixed Links

The layout algorithm does not reshape the links that are specified as fixed. (See *Preserve Fixed Links* on page 32.)

Preserve Fixed Nodes

The layout algorithm does not move the nodes that are specified as fixed. (See *Preserve Fixed Nodes* on page 32.)

Save Parameters to Named Properties

The layout algorithm is able to save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Save Parameters to Named Properties* on page 34 and *Saving Layout Parameters and Preferred Layouts* on page 315.)

Stop Immediately

The layout algorithm stops after cleanup if the method `stopImmediately` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop Immediately* on page 35.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific Parameters

The following parameters are specific to the `IlvCircularLayout` class:

Clustering Mode

The Circular Layout algorithm has two clustering modes. To select a clustering mode, use the method:

```
void setClusteringMode(int mode);
```

To obtain the current clustering mode, call:

```
int getClusteringMode();
```

The valid values for `mode` are:

- `IlvCircularLayout.BY_CLUSTER_IDS` (the default): Cluster identifiers need to be explicitly provided for each node (see *Cluster Membership and Order of the Nodes on a Cluster* on page 245). A tree-like algorithm places the clusters.
- `IlvCircularLayout.BY_SUBGRAPHS`: The algorithm handles a nested graph, including intergraph links. It arranges the nodes of each subgraph on a circle, so that the number of link crossings is small. It respects the intergraph links and rotates the cluster so that the number of link crossings is small. It assumes that all nodes are nearly square and that all nodes are in subgraphs, but the subgraph nesting is only 1. Nodes that are inside subgraphs of subgraphs are not handled. Note that in this mode each subgraph keeps its initial position. The subgraphs can be placed either by a different layout algorithm or interactively.

Cluster Membership and Order of the Nodes on a Cluster

This section applies only if he clustering mode is `BY_SUBGRAPHS`.

Before performing the layout, you must specify to which cluster each node of the graph belongs. To do this, use a cluster identifier; that is, an instance of a subclass of the class `IlvClusterId` (which is abstract). Two subclasses are provided:

- `IlvClusterNumber`, which uses integer numbers as cluster identifiers.
- `IlvClusterName`, which uses string names as cluster identifiers.

You can combine these two types of identifiers as any other subclass of `IlvClusterId`. For example, you can write:

```
// create identifier for first cluster (integer)
IlvClusterNumber clusterId1 = new IlvClusterNumber(1);
// create identifier for second cluster (string)
IlvClusterNumber clusterId2 = new IlvClusterName("R&D network");
```

Then, if `node1` to `node3` belong to the first cluster, you can write:

```
layout.setClusterId(node1, clusterId1);
layout.setClusterId(node2, clusterId1);
layout.setClusterId(node3, clusterId1);
```

Assume `layout` is an instance of `IlvCircularLayout`.

If you want the nodes to be drawn in a special order (for example, `node1` -> `node2` -> `node3`), you should also specify an index (an integer value) for each node:

```
layout.setClusterId(node1, clusterId1, 0);
layout.setClusterId(node2, clusterId1, 1);
layout.setClusterId(node3, clusterId1, 2);
```

Two methods allow you to specify the cluster to which a node belongs:

```
void setClusterId(Object node, IlvClusterId clusterId)
```

```
void addClusterId(Object node, IlvClusterId clusterId)
```

If you call the first method, the node belongs only to the cluster whose identifier is `clusterId`. The second method allows you to specify that a node belongs to more than one cluster.

These methods have another version with an additional argument, an integer value representing the index:

```
void setClusterId(Object node, IlvClusterId clusterId, int index)
```

```
void addClusterId(Object node, IlvClusterId clusterId, int index)
```

This value is used to order the nodes on the cluster. If you specify these indices, the algorithm sorts the nodes in ascending order according to the index values.

Note that the values of the index cannot be negative. They do not need to be continuous; only the order of the values is important.

To obtain the current index of a node on a given cluster, use the method:

```
int getIndex(Object node, IlvClusterId clusterId)
```

If no index is specified for the node, the method returns the value `IlvCircularLayout.NO_INDEX`. It is a negative value.

To obtain an enumeration of the cluster identifiers for the clusters to which the node belongs, use the method:

```
Enumeration getClusterIds(Object node)
```

The elements of the enumeration are instances of a subclass of `IlvClusterId`.

To efficiently obtain the number of clusters to which a node belongs, use the method:

```
int getClusterIdsCount(Object node)
```

To remove a node from a cluster with a given identifier, use the method:

```
void removeClusterId(Object node, IlvClusterId clusterId)
```

To remove a node from all the clusters to which it belongs, use the method:

```
void removeAllClusterIds(Object node)
```

Star Center

To specify whether a node is the center of a star, call the method:

```
void setStarCenter(Object node, boolean starCenter)
```

By default, a node is not the center of a star.

To know whether a node is the center of a star, use the method:

```
boolean isStarCenter(Object node)
```

This parameter has no effect if the clustering mode is `BY_SUBGRAPHS`.

Root Clusters

The algorithm arranges the clusters of each connected component of the graph of clusters around a “root cluster”. By default, the algorithm can choose this cluster. Optionally, you can specify one or more root clusters (one for each connected component), using the methods:

```
void setRootClusterId(IlvClusterId clusterId)
```

To obtain an enumeration of the identifiers of the clusters that have been specified as root clusters, use the method:

```
Enumeration getRootClusterIds()
```

This parameter has no effect if the clustering mode is `BY_SUBGRAPHS`.

Area Minimization

For very large graphs, the radius of the concentric circles on which the clusters are placed can become very large. Therefore, the Circular Layout provides an optional mode that reduces the total area of the layout. To reduce the total area, the clusters are distributed more equally on the circle. To enable or disable the area minimization mode, use the following method:

```
void setAreaMinimizationEnabled(boolean option)
```

The default value is `false` (area minimization is disabled).

Deciding whether to enable the area minimization mode essentially depends on the size of the network. We recommend the area minimization mode for very large networks.

To obtain the current choice, use the method:

```
boolean isAreaMinimizationEnabled()
```

To get an idea of the difference between these modes, compare the following layouts of the same network:

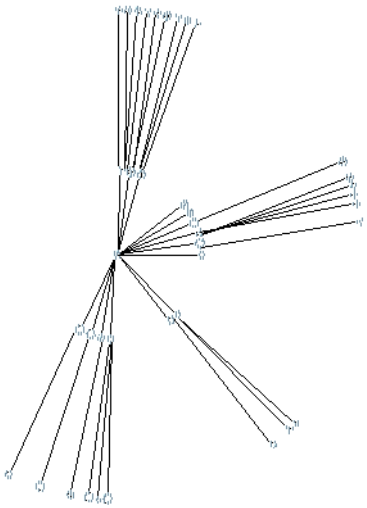


Figure 4.121 *Area Minimization Disabled (default)*

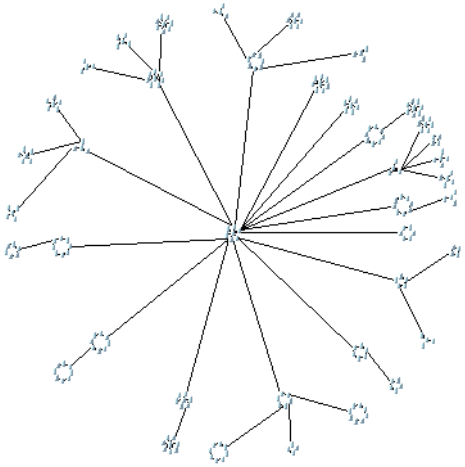


Figure 4.122 *Area Minimization Enabled*

This parameter has no effect if the clustering mode is BY_SUBGRAPHS.

Dimensional Parameters

Figure 4.123 illustrates the dimensional parameters used in the Circular Layout algorithm. These parameters are explained in the sections that follow.

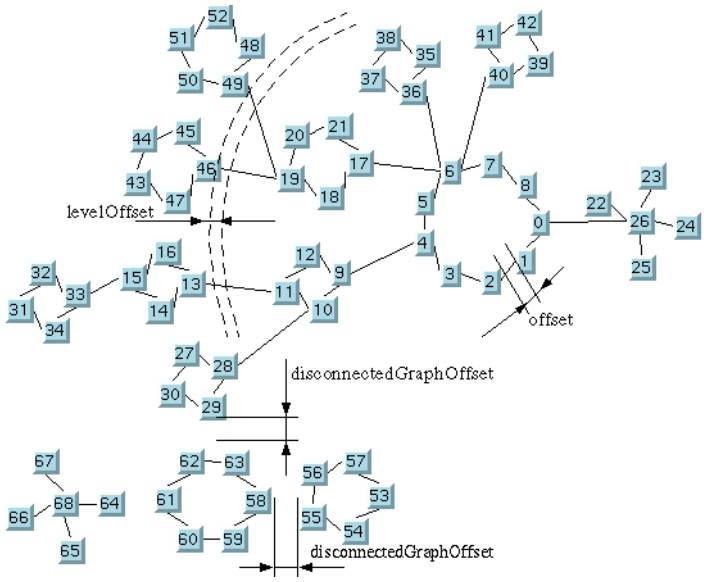


Figure 4.123 *Dimensional Parameters for the Circular Layout Algorithm*

Offset

The layout algorithm tries to preserve a minimum distance between nodes (see Figure 4.123). To specify the offset, use the method:

```
void setOffset(float offset)
```

To obtain the current value, use the method:

```
float getOffset()
```

Level Offset

If the clustering mode is `BY_SUBGRAPHS`, the level offset parameter controls the minimal offset between nodes that belong to the same cluster.

The following applies if the clustering mode is `BY_CLUSTER_IDS`.

As explained in *Brief Description of the Algorithm* on page 240, interconnected rings and/or clusters are drawn on concentric circles around a root cluster. The radius of each concentric circle is computed to avoid overlapping clusters. In some cases, you may want to increase this radius to obtain a clearer drawing of the network. To meet this purpose, the radius is

systematically increased with a "level offset" value (see Figure 4.123). To specify this parameter, use the method:

```
void setLevelOffset(float offset)
```

To obtain the current value, use the method:

```
float getLevelOffset()
```

The default value is zero.

This parameter has no effect if the clustering mode is BY_SUBGRAPHS.

Disconnected Graph Offset

As explained in *Brief Description of the Algorithm* on page 240, each connected component of the network is laid out separately and the drawing of each component is placed near the others (see Figure 4.123). To specify the offset between each connected component, use the method:

```
void setDisconnectedGraphOffset(float offset)
```

To obtain the current value, use the method:

```
float getDisconnectedGraphOffset()
```

This parameter has no effect if the clustering mode is BY_SUBGRAPHS.

Get the Contents, the Position, and the Size of the Clusters

At times, you might need to know the position and the size of the circle on which the nodes for each cluster are drawn. This may be the case if you want to perform some reshaping operations on the links. To do this, you can obtain a vector containing all the cluster identifiers after the layout is performed. Use the following method:

```
Vector getClusterIds()
```

The vector contains instances of a subclass of `IlvClusterId`. By browsing the elements of this `Vector`, you can get the necessary information for each cluster:

```
float getClusterRadius(int clusterIndex)

IlvPoint getClusterCenter(int clusterIndex)

Vector getClusterNodes(int clusterIndex)
```

The `getClusterNodes` method returns the nodes that make up the cluster. The argument `clusterIndex` represents the position of the cluster in the `Vector` returned by the method `getClusterIds()`.

These methods should not be used if the clustering mode is BY_SUBGRAPHS.

Link Style

When the layout algorithm moves the nodes, straight-line links, such as instances of `IlvLinkImage`, will automatically "follow" the new positions of their end nodes. If the grapher contains other types of links (for example, `IlvPolylineLinkImage` or `IlvSplineLinkImage`), the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can ask the layout algorithm to automatically remove all the intermediate points of the links (if any). To do this, the following method is provided:

```
void setLinkStyle(int style)
```

The valid values for `style` are:

- `IlvCircularLayout.NO_RESHAPE_STYLE`

  None of the links is reshaped in any manner.

- `IlvCircularLayout.STRAIGHT_LINE_STYLE`

  All the intermediate points of the links (if any) are removed. This is the default value.

> ***Note:*** *If the* `STRAIGHT_LINE_STYLE` *is selected, the layout algorithm may raise an* `IlvInappropriateLinkException` *if layout is performed on an* `IlvGrapher`*, but links that cannot be given a straight-line shape are used. Such links are* `IlvOneLinkImage`*,* `IlvDoubleLinkImage`*,* `IlvOneSplineLinkImage`*, and* `IlvDoubleSplineLinkImage`*. You can use the methods* `EnsureAppropriateLinkTypes` *or* `EnsureAppropriateLinks` *defined in* `IlvGraphLayoutUtil` *to automatically replace inappropriate links. This can be done before layout or when the* `IlvInappropriateLinkException` *is caught. For details on these methods, see the ILOG JViews Graph Layout Reference Manual. For details on the graph model, see Using the Graph Model on page 297.*

To obtain the current choice, use the method:

```
int getLinkStyle()
```

Using a Link Clipping Interface

By default, the Circular Layout does not place the connection points of links. It relies on the link connectors of the nodes to determine the connection points. If no link connectors are installed at the nodes, the default behavior is to connect to a point at the border of the bounding box of the nodes. If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape. An example is shown in Figure 4.124.

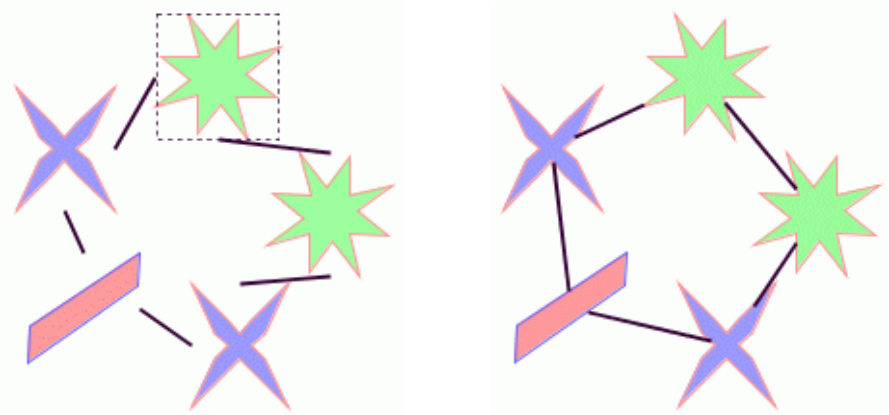


Figure 4.124 *Effect of Link Clipping Interface*

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkClipInterface`. An example for the implementation of a link clip interface is in *Link Clipping* on page 27. To set a link clip interface, call:

```
void setLinkClipInterface(IlvLinkClipInterface interface)
```

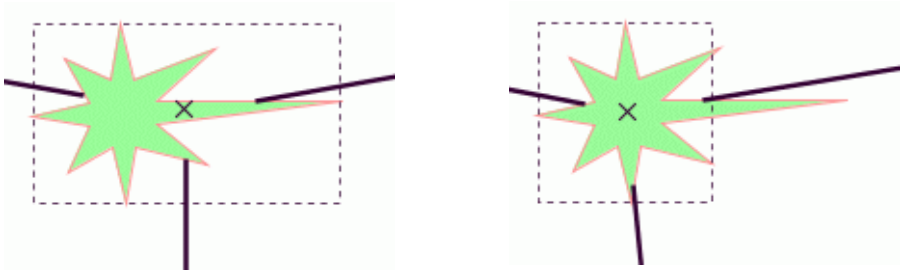
> ***Note:*** *The link clip interface requires link connectors at the nodes of an* `IlvGrapher` *that allow connector pins to be placed freely at the node border. It is recommended that you use* `IlvRelativeLinkConnector` *or* `IlvClipLinkConnector` *for link connectors to be used in combination with* `IlvGrapher` *objects. The clip link connector updates the clipped connection points automatically during interactive node movements.*

If a node has an irregular shape, the clipped links sometimes should not point towards the center of the node bounding box, but to a virtual center inside the node. You can achieve this by additionally providing a class that implements the `IlvLinkConnectionBoxInterface`. An example for the implementation of a link connection box interface is in*Link Connection Box* on page 29. To set a link connection box interface, call:

```
void setLinkConnectionBoxInterface(IlvLinkConnectionBoxInterface
interface)
```

The link connection box interface is used only when link clipping is enabled by setting a link clip interface. If no link clip interface is specified, the link connection box interface has no effect.

An example of the combined effect is shown in Figure 4.125.



Clipping at the node bounding box

Clipping at a specified connection box

Figure 4.125 *Combined Effect of Link Clipping Interface and Link Connection Box*

If the links are clipped at the green irregular star node (Figure 4.125 left), they do not point towards the center of the star, but towards the center of the bounding box of the node. This can be corrected by specifying a link connection box interface that returns a smaller node box than the bounding box (Figure 4.125 right). Alternatively, the problem could be corrected by specifying a link connection box interface that returns the bounding box as the node box but with additional tangential offsets that shift the virtual center of the node.

Grid Layout

In this section, you will learn about the *Grid Layout* algorithm from the ILOG JViews Graph Layout module (class `IlvGridLayout` from the package `ilog.views.graphlayout.grid`).

Sample

Here are sample drawings produced with the Grid Layout:

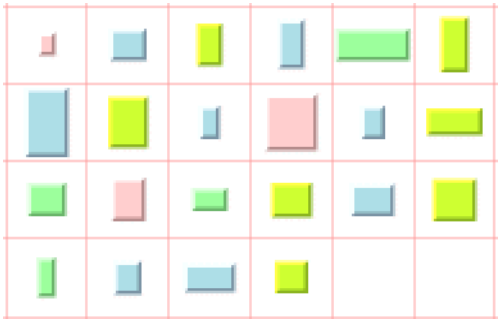


Figure 4.126 *Drawing produced using the* `TILE_TO_GRID_FIXED_WIDTH` *mode with* `CENTER` *horizontal and vertical alignment. The red lines are drawn here to help the identification of the grid cells; they are not drawn by the layout algorithm.*

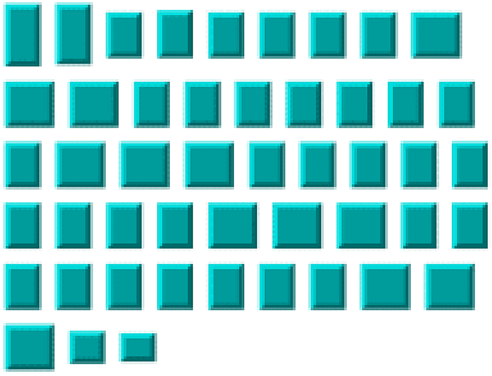


Figure 4.127 *Drawing produced using the* `TILE_TO_ROWS` *mode with* `CENTER` *vertical alignment.*

What Types of Graphs?

- Any graph. However, the links are never taken into consideration. This algorithm is designed for placing nodes independently on their eventual links.

Application Domains

Any domain where a collection of isolated nodes needs to be laid out.

Features

- Arranges a collection of isolated nodes or connected components.
- Takes into account the size of the nodes so that no overlapping occurs.
- Provides several alignment options and dimensional parameters.
- Provides full support for fixed nodes (overlapping of nonfixed nodes with fixed nodes is avoided).
- Provides an incremental mode which helps the retention of a mental map on incremental changes made to a collection of nodes.

Brief Description of the Algorithm

The Grid layout has two main modes: *grid* and *row/column*.

- In grid mode, the layout arranges the nodes of a graph in the cells of a grid (matrix). If a node is too large to fit in one grid cell (with margins), it occupies multiple cells. The size of the grid cells and the margins are parameters of the algorithm.
- In row/column mode, the layout arranges the nodes of a graph either by rows or by columns (according to the specified option). The width of the rows is controlled by the width of the layout region parameter. The height of the columns is controlled by the height of the layout region parameter. The horizontal and vertical margins between the nodes are parameters of the algorithm.

Code Sample

Below is a code sample using the `IlvGridLayout` class:

```
 ...
import ilog.views.*;
import ilog.views.graphic.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.grid.*;
 ...
IlvGrapher grapher = new IlvGrapher();
```

```
IlvManagerView view = new IlvManagerView(grapher);

 ... /*  Fill in the grapher with nodes and links here */

IlvGridLayout layout = new IlvGridLayout();
layout.attach(grapher);

try {
        IlvGraphLayoutReport layoutReport = layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Generic Features and Parameters

The `IlvGridLayout` class supports the following generic parameters defined in the `IlvGraphLayout` class:

- *Allowed Time*
- *Layout Region*
- *Preserve Fixed Nodes*
- *Save Parameters to Named Properties*
- *Stop Immediately*

The following comments describe the particular way in which these parameters are used by this subclass.

Allowed Time

The layout algorithm stops if the allowed time setting has elapsed. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Allowed Time* on page 22.) The result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Layout Region

The layout algorithm uses the layout region setting (either your own or the default setting) to control the size and the position of the graph drawing. All three ways to specify the layout region are available for this subclass. (See *Layout Region* on page 25.)

The layout region is considered differently depending on the layout mode. For details, see *Layout Modes* on page 258.)

Preserve Fixed Nodes

The layout algorithm does not move the nodes that are specified as fixed. (See *Preserve Fixed Nodes* on page 32.) Moreover, nonfixed nodes are placed in such a manner that overlaps with fixed nodes are avoided.

Save Parameters to Named Properties

The layout algorithm can save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Save Parameters to Named Properties* on page 34 and *Saving Layout Parameters and Preferred Layouts* on page 315.)

Stop Immediately

The layout algorithm stops after cleanup if the method `stopImmediately` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop Immediately* on page 35.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific Parameters

The following parameters are specific to the `IlvGridLayout` class.

Layout Modes

The Grid Layout algorithm has four layout modes. To select a layout mode, use the method:

```
void setLayoutMode(int mode);
```

To obtain the current layout mode, call:

```
int getLayoutMode();
```

The valid values for `mode` are:

- `IlvGridLayout.TILE_TO_GRID_FIXED_WIDTH` (the default).

  The nodes are placed in the cells of a grid (matrix) that has a fixed maximum number of columns. This number is equal to the width of the layout region parameter divided by the horizontal grid offset.

- `IlvGridLayout.TILE_TO_GRID_FIXED_HEIGHT`

  The nodes are placed in the cells of a grid (matrix) that has a fixed maximum number of rows. This number is equal to the height of the layout region parameter divided by the vertical grid offset.

- `IlvGridLayout.TILE_TO_ROWS`

  The nodes are placed in rows. The maximum width of the rows is equal to the width of the layout region parameter. The height of the row is the maximum height of the nodes contained in the row (plus margins).

- `IlvGridLayout.TILE_TO_COLUMNS`

  The nodes are placed in columns. The maximum height of the columns is equal to the height of the layout region parameter. The width of the column is the maximum width of the nodes contained in the column (plus margins).

Alignment

The alignment options control how a node is placed over its grid cell or over its row or column (depending on the layout mode). The alignment can be set globally, in which case all nodes are aligned in the same way, or locally on each node, with the result that different alignments occur in the same drawing.

Global Alignment Parameters

To set and retrieve the global alignment, use the following methods:

```
void setGlobalHorizontalAlignment(int alignment);

void setGlobalVerticalAlignment(int alignment);

int getGlobalHorizontalAlignment();

int getGlobalVerticalAlignment();
```

The valid values for `alignment` are:

- `IlvGridLayout.CENTER` (the default)

  The node is horizontally and/or vertically centered over its grid cell or row or column.

- `IlvGridLayout.TOP`

  The node is vertically aligned on the top of its cell(s) or row. Not used if the layout mode is `TILE_TO_COLUMNS`.

- `IlvGridLayout.BOTTOM`

  The node is vertically aligned on the bottom of its grid cell(s) or row. Not used if the layout mode is `TILE_TO_COLUMNS`.

- `IlvGridLayout.LEFT`

  The node is horizontally aligned on the left of its grid cell(s) or column. Not used if the layout mode is `TILE_TO_ROWS`.

- `IlvGridLayout.RIGHT`

  The node is horizontally aligned on the right of its grid cell(s) or column. Not used if the layout mode is `TILE_TO_ROWS`.

- `IlvGridLayout.MIXED`

  Each node can have a different alignment. The alignment of each individual node can be set with the result that different alignments can occur in the same graph.

Alignment of Individual Nodes

All nodes have the same alignment unless the global alignment is set to `IlvGridLayout.MIXED`. Only when the global alignment is set to mixed can each node have an individual alignment style.

To set and retrieve the alignment of an individual node, use the following methods:

```
void setHorizontalAlignment(Object node, int alignment);

void setVerticalAlignment(Object node, int alignment);

int getHorizontalAlignment(Object node);

int getVerticalAlignment(Object node);
```

The valid values for `alignment` are:

- `IlvGridLayout.CENTER` (the default)
- `IlvGridLayout.TOP`
- `IlvGridLayout.BOTTOM`
- `IlvGridLayout.LEFT`
- `IlvGridLayout.RIGHT`

Maximum Number of Nodes Per Row or Column

By default, in `IlvGridLayout.TILE_TO_ROWS` or `IlvGridLayout.TILE_TO_COLUMNS` mode, the layout places as many nodes on each row or column as possible given the size of the nodes and the dimensional parameters (layout region and margins). If needed, the layout can additionally respect a specified maximum number of nodes per row or column.

To set and retrieve the maximum number of nodes per row or column, use the methods:

```
void setMaxNumberOfNodesPerRowOrColumn(int nNodes);

int getMaxNumberOfNodesPerRowOrColumn();
```

The default value is `Integer.MAX_VALUE`, that is, the number of nodes placed in each row or column is bounded only by the size of the nodes and the dimensional parameters. The specified value must be at least `1`. The parameter has no effect if the layout mode is `IlvGridLayout.TILE_TO_GRID_FIXED_WIDTH` or `IlvGridLayout.TILE_TO_GRID_FIXED_HEIGHT`.

Incremental Mode

The Grid Layout algorithm normally places all the nodes from scratch. If the graph incrementally changes because you add, remove, or resize nodes, the subsequent layout may differ considerably from the previous layout. To avoid this effect and to help the user to retain a mental map of the graph, the algorithm has an incremental mode. In incremental mode, the layout tries to place the nodes at the same location or in the same order as in the previous layout whenever it is possible.

To enable the incremental mode, call the method `setIncrementalMode` as follows:

```
layout.setIncrementalMode(true);
```

> ***Note:*** *To preserve the stability, the incremental mode can keep some regions free. Therefore, the total area of the layout can be larger than in nonincremental mode, and, in general, the layout may not look as nice as in nonincremental mode.*

Dimensional Parameters

Figure 4.128 and Figure 4.129 illustrate the dimensional parameters used in the Grid Layout algorithm. These parameters are explained in the sections that follow.

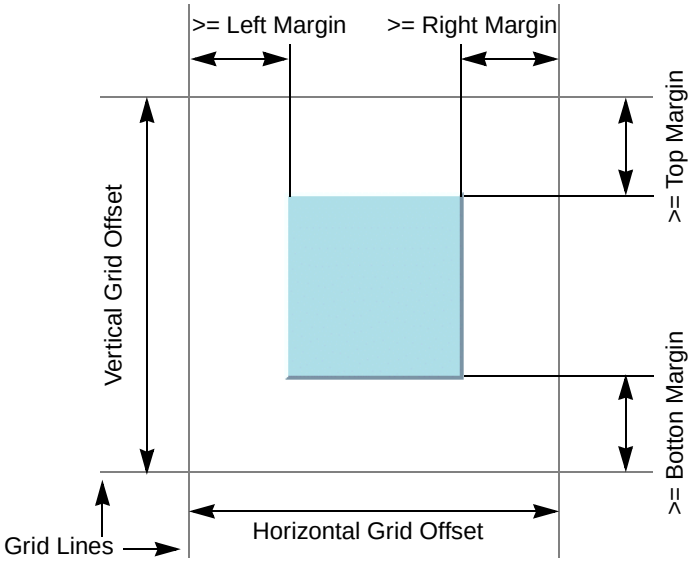


Figure 4.128 *Dimensional Parameters for the Grid Mode of the Grid Layout Algorithm*

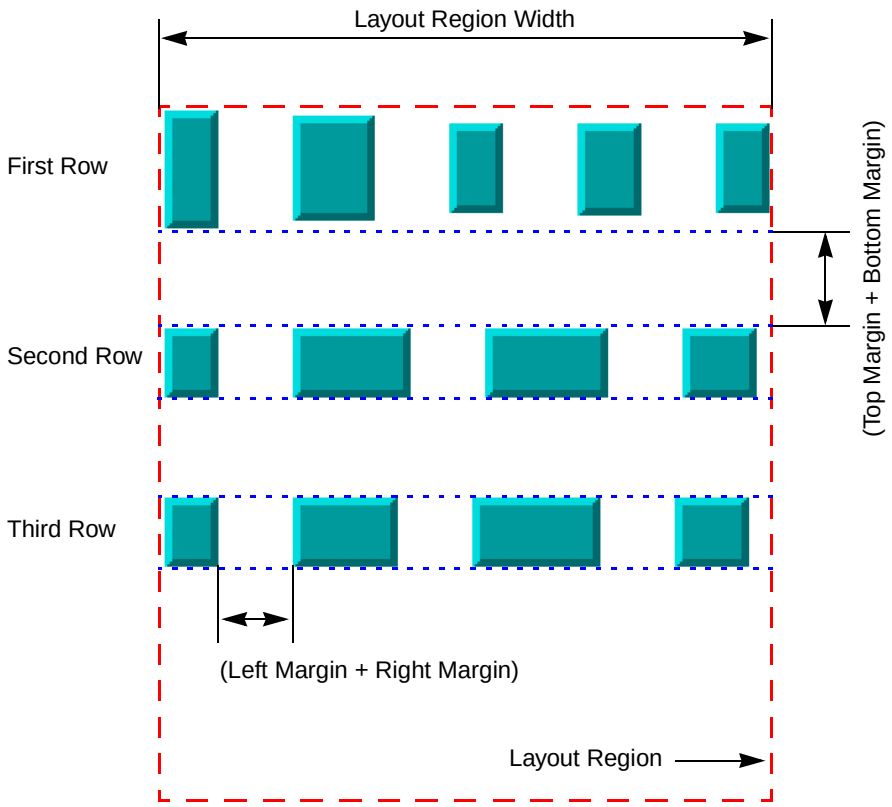


Figure 4.129 *Dimensional Parameters for the Row/Column Mode of the Grid Layout Algorithm*

Grid Offset

The grid offset parameters control the spacing between grid lines. It is taken into account only by the grid mode (layout modes `TILE_TO_GRID_FIXED_WIDTH` and `TILE_TO_GRID_FIXED_HEIGHT`). The horizontal and vertical grid offset can be set and retrieved by the following methods:

```
void setHorizontalGridOffset(float offset);

void setVerticalGridOffset(float offset);

float getHorizontalGridOffset();

float getVerticalGridOffset();
```

The grid offset is the critical parameter for the grid mode. If the grid offset is larger than the size of the nodes (plus margins), an empty space is left around the node. If the grid offset is smaller than the size of the nodes (plus margins), the node will need to be placed on more

than one grid cell. The best choice for the grid offsets depends on the application. It can be computed according to either the maximum size of the nodes (plus margins) or the medium size, and so on. Of course, if all the nodes have a similar size, the choice is straight-forward.

Margins

The margins control the space around each node that the layout algorithm keeps empty. The margins can be set and retrieved using the following methods:

```
void setTopMargin(float margin);

void setBottomMargin(float margin);

void setLeftMargin(float margin);

void setRightMargin(float margin);

float getTopMargin();

float getBottomMargin();

float getLeftMargin();

float getRightMargin();
```

The meaning of the margin parameters is not the same for the grid modes as for the row/column modes. In the grid modes, they represent the minimum distance between the node border and the grid line (see Figure 4.128 on page 261.) In the row/column modes, they are used to control the vertical distance between the rows or the horizontal distance between the columns and the horizontal or vertical minimal distance between the nodes in the same row or column (see Figure 4.129 on page 262).

The default value for all the margin parameters is `5`.

Sequences of Graph Layout

In some circumstances, you may need to use a sequence of layouts on the same graph. For example:

- You work with graphs that have become out-of-date and need to extend the graph. If you perform a layout on the extended graph, you probably want to identify the parts that were already laid out in the original graph. The layout should not change very much when compared to the layout of the original graph.
- The first layout results in a drawing with minor deficiencies. You want to solve these deficiencies manually and perform a second layout to clean up the drawing. The second layout probably should not change very much the parts of the graph that were already acceptable after the first layout.

The Hierarchical Layout supports sequences of layout that “do not change very much”. It allows you to preserve the level index of a node and the position index of the node within the level. By doing this, the relative position of a node compared to a previous layout does not

change. The algorithm calculates new coordinates for the nodes and new routings for the links to adjust the absolute positions of the objects in order to clean up the drawing. The relative order such as “node A is in a level above node B” or “node A is left of node B” can be preserved if you use the combination of specified and calculated level index and position index. This is described in the following two sections.

Recursive Layout

In this section, you will learn about the *Recursive Layout* class from the ILOG JViews Graph Layout module (class `IlvRecursiveLayout` from the package `ilog.views.graphlayout.recursive`).

Application Domain

The `IlvGrapher` class provided by the ILOG JViews Graphic Framework allows nested graphs to be represented (for details see the chapter *Nested Managers and Nested Graphers* in the *ILOG JViews Graphics Framework User's Manual*). An example of a nested graph is given in Figure 4.130.

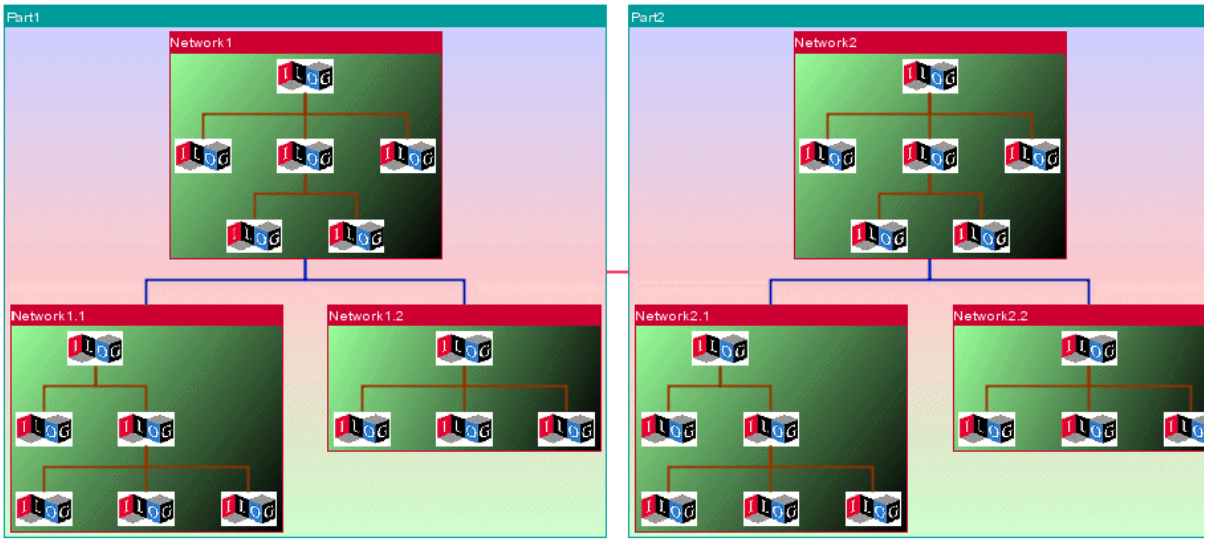


Figure 4.130 *Example of a Recursive Layout of a Nested Graph*

In this example, there is a topmost graph which contains only two nodes, named `Part1` and `Part2`, connected by a link. Each of these two nodes is another graph, composed of three nodes connected by links. Finally, each of these nodes is another graph, composed of some icons connected by links.

Such nested graphs are often used to represent the hierarchical decomposition of a network into subnetworks, more generally to mix connectivity information for several levels of detail in the same drawing.

In Figure 4.130, a layout algorithm (the Tree Layout) has been used for each of the graphs.

The Recursive Layout class is not a layout algorithm but rather a facility to apply another layout algorithm recursively on a nested graph. It traverses the nesting structure starting from the graph that is attached to the Recursive Layout itself and recursively applies a layout on all subgraphs. You can tailor which sublayout must be applied to which subgraph.

There are basically two scenarios:

- The same layout style must be applied to all subgraphs.
- An individual layout style must be applied to each subgraph.

Code Sample: Same Layout Style Everywhere

First, we assume you want to apply a Tree Layout to a nested graph. Each subgraph should be laid out with the same global layout parameters. The Tree Layout algorithm itself handles only flat graphs, that is, if applied to an attached graph, it lays out only the nodes and links of the attached graph, but not the nodes and links of the subgraphs that are nested inside the attached graph. Hence, we encapsulate the Tree Layout into a Recursive Layout. The Recursive Layout traverses the entire nesting hierarchy of the attached graph, while the encapsulated Tree Layout lays out each (flat) subgraph of the nesting hierarchy during the traversal.

```
...
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.recursive.*;
import ilog.views.graphlayout.tree.*;

IlvRecursiveLayout layout = new IlvRecursiveLayout(IlvTreeLayout());

IlvGrapher topLevelGrapher = ...
layout.attach(topLevelGrapher);
try {
        IlvRecursiveLayoutReport layoutReport =
              (IlvRecursiveLayoutReport)layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
...
// detach the Recursive Layout when it is no longer needed
layout.detach();
...
```

This mode of the Recursive Layout is called *reference layout mode*. In this case, a Tree Layout is performed recursively to the top-level graph and to each subgraph. All layouts are performed with the same global layout parameters.

> ***Note:*** *Here we use the term "global layout parameter" for the parameters that do not depend on a specific node or link. For instance, Tree Layout has a global layout parameter set by* `setGlobalLinkStyle`*, as well as a layout parameter set by* `setLinkStyle(link, style)` *which is local to a link.*

You can change the global layout parameters by accessing the reference layout of the Recursive Layout:

```
IlvTreeLayout treeLayout = (IlvTreeLayout)layout.getReferenceLayout();
treeLayout.setFlowDirection(IlvDirection.Left);
```

Technically, the reference layout instance is not applied to each subgraph because each subgraph needs an individual layout instance. The reference layout instance is only applied to the top-level graph. Furthermore, a clone of the reference instance is created for each subgraph. This clone remains attached to the subgraph as long as the Recursive Layout is attached to the top-level graph. Before layout is performed, the global layout parameters are copied from the reference layout instance to each cloned layout instance.

Sometimes, you want to specify local layout parameters for individual nodes and links. In this case, you need to access the cloned layout instance that is attached to the subgraph that owns the node or link. For instance, to the the link style of an individual link, use:

```
IlvTreeLayout treeLayout = (IlvTreeLayout)layout.getLayout(link.getGraphicBag());
treeLayout.setLinkStyle(link, IlvTreeLayout.ORTHOGONAL_STYLE);
```

You cannot use the reference layout mode in the following cases:

- The layout algorithm to be applied on subgraphs is not the same as the algorithm needed for the topmost graph (the reference layout).
- The same layout algorithm, but using different global parameter settings, needs to be applied on different subgraphs.

In this case, you can use one of the following modes.

Code Sample: Mixing Different Layout Styles

The following example shows the second scenario: Each subgraph should be laid out by a different layout style or with individual global layout parameters. In this case, you use the *internal provider mode* of the Recursive Layout.

We assume that you have a graph with three subgraphs. The top-level graph and the first subgraph should be processed with Tree Layout, the second subgraph with Bus Layout, and

the third subgraph with Grid Layout. You have to specify which layout should be used for which subgraph, and then you can perform the layout.

```
...
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.recursive.*;
import ilog.views.graphlayout.tree.*;
import ilog.views.graphlayout.bus.*;
import ilog.views.graphlayout.grid.*;

IlvRecursiveLayout layout = new IlvRecursiveLayout();
IlvGrapher topLevelGrapher = ...
layout.attach(topLevelGrapher);

// specify the layout of the top level graph
layout.setLayout(null, new IlvTreeLayout());
// specify the layout of subgraphs
layout.setLayout(subgraph1, new IlvTreeLayout());
layout.setLayout(subgraph2, new IlvBusLayout());
layout.setLayout(subgraph3, new IlvGridLayout());

// perform layout
try {
        IlvRecursiveLayoutReport layoutReport =
              (IlvRecursiveLayoutReport)layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
...
// detach the Recursive Layout when it is no longer needed
layout.detach();
...
```

In this scenario, there is no reference layout. All layout parameters of different subgraphs are independent. You need to specify new, independent layout instances for each subgraph; otherwise no layout will be performed for the corresponding subgraph. The layout instances are attached to the subgraph as long as the Recursive Layout is attached to the top-level graph. You can specify in this example different global layout parameters for the Tree Layout of the top-level graph and the Tree Layout of `subgraph1`. You access the layout instance of each individual subgraph in order to change global layout parameters for this subgraph as well as parameters of nodes and links of the subgraph. For instance if `node1` belongs to `subgraph1` and `node2` belongs to `subgraph2`, you can set individual global and local layout parameters in this way:

```
// access the layout of the top level graph
IlvTreeLayout treeLayout1 = (IlvTreeLayout)layout.getLayout(null);
treeLayout1.setFlowDirection(IlvDirection.Bottom);

// access the layouts of the subgraphs
IlvTreeLayout treeLayout2 = (IlvTreeLayout)layout.getLayout(subgraph1);
treeLayout2.setFlowDirection(IlvDirection.Left);
treeLayout2.setAlignment(node1, IlvTreeLayout.TIP_OVER);

IlvBusLayout busLayout = (IlvBusLayout)layout.getLayout(subgraph2);
busLayout.setOrdering(IlvBusLayout.ORDER_BY_HEIGHT);
busLayout.setBus(node2);

IlvGridLayout gridLayout = (IlvGridLayout)layout.getLayout(subgraph3);
gridLayout.setLayoutMode(IlvGridLayout.TILE_TO_COLUMNS);
```

Code Sample: Using a Specified Layout Provider

The ILOG JViews Graph Layout library provides a flexible mechanism for the choice of the layout instance to be applied to each subgraph in a nested graph: the layout provider. In the previous example, an layout provider was used internally. For simplicity, the details of the mechanism are hidden, and you select the choice of layout by using the method `setLayout` on the Recursive Layout instance. Therefore, this layout mode is called *internal provider mode*.

However, you can also design your own layout provider and use it inside the Recursive Layout. This is the *specified provider mode* of the Recursive Layout.

A layout provider is a class that implements the interface `IlvLayoutProvider`. The interface has a unique method:

```
getGraphLayout(IlvGraphModel graphModel)
```

This method must return the layout instance to be used for the graph model passed as the argument, or `null` if no layout is required for this graph. When performing the Recursive Layout, these methods get the layout instance to be used for each graph from the specified layout provider.

To implement the interface `IlvLayoutProvider`, you must decide how the choice of the layout instance is done. This can be based on some criteria such as the type of graph (eventually known in advance), or a choice already made by the end user and recorded, for instance, in a property of the graph. A possible implementation of the `getGraphLayout` method is the following:

```
public IlvGraphLayout getGraphLayout(IlvGraphModel graphModel)
{
  Object prop = graphModel.getProperty("layout type");
  // if none, return null (no layout needed for this graph)
  if (!(prop instanceof String))
    return null;

  IlvGraphLayout layout = null;
```

```
  String name = (String)prop;

  if (name.equals("tree"))
    layout = new IlvTreeLayout();
  else if (name.equals("flow"))
    layout = new IlvHierarchicalLayout();
  else
    throw new RuntimeException("unsupported layout choice: " + name);

  layout.attach(graphModel);

  return layout;
}
```

Of course, this is only an example among many possible implementations. The implementation may decide to store the newly allocated layout instance to avoid allocating a new one when the method is again called for the same graph.

If you have implemented a layout provider, you can use it in the Recursive Layout in the following way:

```
..
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.recursive.*;

IlvLayoutProvider layoutProvider = ...
IlvRecursiveLayout layout = new IlvRecursiveLayout(layoutProvider);
IlvGrapher topLevelGrapher = ...
layout.attach(topLevelGrapher);

// Perform the layout
try {
        IlvRecursiveLayoutReport layoutReport =
              (IlvRecursiveLayoutReport)layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
...
// detach the Recursive Layout when it is no longer needed
layout.detach();
...
```

Features

- This subclass of `IlvGraphLayout` is not a usual layout algorithm but rather a facility to manage the layout of a nested grapher.
- Three layout modes: reference layout mode, internal provider mode, and specified provider mode.
- Allows you to perform a layout algorithm recursively in a nested grapher.
- Allows you to perform a recursive layout on a nested grapher while each subgrapher uses an individual layout style.
- Layout features, speed, and quality depend on the features, speed, and quality of the sublayouts.

Generic Features and Parameters

Depending on the support of its sublayouts, Recursive Layout may support the following generic parameters defined in the `IlvGraphLayout` class:

- Allowed Time
- Percentage Completion Calculation
- Save Parameters to Named Properties
- Stop Immediately

The following paragraphs describe the particular way in which these parameters are used by this subclass.

Allowed Time

The Recursive Layout can stop the entire layout of a nested graph after a certain amount of time. If the allowed time setting has elapsed, the Recursive Layout stops; that means it stops the currently running layout of a subgraph and skips the subsequent layouts of subgraphs that have not yet been started. If at the stop time point a sublayout is running on a subgraph that does not support the “allowed time” feature, then this sublayout first runs to completion before the Recursive Layout is stopped. If the Recursive Layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Percentage Completion Calculation

The Recursive Layout calculates the percentage of completion. This value can be obtained from the layout report during the run of the layout. The value is, however, a very rough estimation. If the layouts on the subgraphs do not support the calculation of the percentage completion, the Recursive Layout can report the percentage based only on the information how many layouts of subgraphs are already finished. For instance, if the entire nesting

structure contains five nested graphs, the mechanism reports 20% after the layout of the first subgraph has finished, 40% after the layout of the second subgraph has finished, and so on. If the layouts of the subgraphs support the calculation of the percentage completion, the Recursive Layout calculates a more detailed percentage. In most cases, the calculated percentage is only a very rough estimation that does not always grow linearly over time. (For a detailed description of this feature, see *Percentage Completion Calculation* on page 31 and *Layout Listeners* on page 279.)

Save Parameters to Named Properties

The Recursive Layout instance can save its layout parameters into named properties if all its sublayouts support this feature. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Save Parameters to Named Properties* on page 34 and *Saving Layout Parameters and Preferred Layouts* on page 315.)

Stop Immediately

The Recursive Layout can be stopped at any time. It stops the currently running layout of a subgraph after cleanup if the method `stopImmediately` is called and skips the subsequent layouts of subgraphs that have not yet been started. If at the stop time point a sublayout is running on a subgraph that does not support the "stop immediately" feature, then this sublayout first runs to completion before the Recursive Layout is stopped. (For a description of this method in the `IlvGraphLayout` class, see *Stop Immediately* on page 35.) If the layout stops before completion, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Layout modes

The Recursive Layout has three different layout modes. The layout mode is determined by the constructor that you use. The way how to set global layout parameters of the sublayouts that are applied to the subgraphs is slightly different for each layout mode. You can query the current layout mode by using

```
int getLayoutMode()
```

The possible return values are:

- `IlvRecursiveLayout.REFERENCE_LAYOUT_MODE`: The same layout style with the same global layout parameters is applied to all subgraphs of the nested graph.
- `IlvRecursiveLayout.INTERNAL_PROVIDER_MODE`: The layout is applied using an internal recursive layout provider. The layout styles of individual subgraphs can be specified by using the method `setLayout`.
- `IlvRecursiveLayout.SPECIFIED_PROVIDER_MODE`: The layout is applied using an explicitely specified layout provider.

Reference Layout Mode

This mode should be used if you want to apply the same layout style with the same global layout parameters to the entire nested graph. You first need to allocate the reference layout, that is a new instance of any graph layout algorithm (except `IlvRecursiveLayout`) that should be applied to all subgraphs of the nested graph. Then you allocate the Recursive Layout using the constructor with the reference layout as argument

```
IlvRecursiveLayout(IlvGraphLayout referenceLayout)
```

The reference layout is internally actually only used for the top-level graph of the nested graph. Clones of the reference layout are used for the subgraphs. Hence, all subgraphs are laid out with the same global layout parameters. In order to change the global layout parameters, you can access the reference layout by

```
IlvGraphLayout getReferenceLayout()
```

Global layout parameters are those parameters that are independent from specific nodes or links. Other layout parameters are local to specific nodes or links. For instance, in `IlvHierarchicalLayout`, the method `setGlobalLinkStyle(style)` is a global layout parameter, while the method `setLinkStyle(link,style)` is a local layout parameter.

If you need to set layout parameters that are local to an individual node or link, you need to access the particular clone of the reference layout that is responsible for the subgraph that owns the node or link. After attaching the Recursive Layout to the top-level grapher or graph model, you can retrieve the layout instance of a specific subgraph by

```
IlvGraphLayout getLayout(Object subgraph)
```

However, in reference layout mode, it makes no sense to modify any global layout parameter on the returned instance. The global layout parameters are always taken from the reference layout only. If you pass `null` as subgraph, you get the layout instance of the top-level graph. This is actually the same layout instance as the reference layout.

The reference layout and its clones used during recursive layout remain attached to the subgraphs (or the graph models of the subgraphs, respectively) as long as the Reference Layout itself is attached. When detaching the Reference Layout, all layouts of subgraphs are automatically detached as well.

Internal Provider Mode

This mode should be used if you want to perform graph layout on a nested graph, but either you need individual global layout parameters for specific subgraphs, or you want to layout different subgraphs by different styles. In this case, there is no reference layout. You allocate the Recursive Layout using the constructor with no arguments

```
IlvRecursiveLayout()
```

Before you can perform a layout, you need to specify which layout is used for which subgraph. First, you should attach the Recursive Layout to a graph. Then, in order to specify the layout of the top-level graph, call:

```
recursiveLayout.setLayout(null, sublayout);
```

In order to specify the layout of a specific subgraph, call

```
recursiveLayout.setLayout(subgraph, sublayout);
```

It is important that you assign a different layout instance for each subgraph. You cannot share the same layout instance among different subgraphs. We recommend, that you allocate a new, fresh layout instance for each subgraph. If you pass `null` as sublayout, then no layout is performed for this particular subgraph.

To set the layout for a subgraph and recursively for all its subgraphs, you can use

```
setLayout(Object subgraph, IlvGraphLayout layout,
boolean traverse)
```

and pass the `true` argument for the `traverse` flag. This sets the layouts to a clone of the input layout for each subgraph starting at the input subgraph.

Internally, the Recursive Layout uses a layout provider of type `IlvRecursiveLayoutProvider`. You can access the current layout provider by

```
IlvLayoutProvider getLayoutProvider()
```

However, in internal provider mode, it is mostly not necessary to manipulate the layout provider directly.

Notice: since there is no reference layout, global layout parameters are independent for each subgraph. Global and local layout parameters can be set by accessing the particular layout instance that is assigned to a specific subgraph. After attaching the Recursive Layout to the top-level grapher or graph model, you can retrieve the layout instance of a specific subgraph by

```
IlvGraphLayout getLayout(Object subgraph)
```

If you pass `null` as subgraph, you get the layout instance of the top-level graph.

The layout instances of the subgraphs used during recursive layout remain attached to the subgraphs (or the graph models of the subgraphs, respectively) as long as the Reference Layout itself is attached. When detaching the Reference Layout, all layouts of subgraphs are automatically detached as well.

Specified Provider Mode

Like the internal provider mode, the specified provider mode can also be used if you want to perform graph layout on a nested graph, but either you need individual global layout parameters for specific subgraphs, or you want to layout different subgraphs by different styles. However unlike the internal provider mode, it is your own responsibility to manage the specified layout provider. This is probably only necessary in very advanced applications.

In specified provider mode, there is no reference layout. You allocate the Recursive Layout using the constructor with your layout provider as argument

```
IlvRecursiveLayout(IlvLayoutProvider specifiedProvider)
```

You can access the current layout provider by

```
IlvLayoutProvider getLayoutProvider()
```

You should implement your layout provider in a way so that it delivers a different layout instance for each subgraph. The delivered layout instance must be attached to the graph model of the corresponding subgraph.

Notice: since there is no reference layout, global layout parameters are independent for each subgraph. It is recommended that the implementation of the layout provider takes care of the setting of global and local layout parameters. Theoretically, you can use the method

```
IlvGraphLayout getLayout(Object subgraph)
```

which will return the layout instance that the specified layout provider delivers for the graph model of the input subgraph. If you pass `null` as subgraph, you get the layout instance of the top-level graph. However, the effect of this method depends on the implementation of the layout provider that was passed to the constructor of Recursive Layout.

The layout instances of the subgraphs used during recursive layout should be attached by the layout provider. They are usually not automatically detached when the Recursive Layout is detached. Unless you use one of the predefined providers of class `IlvDefaultLayoutProvider` or `IlvRecursiveLayoutProvider`, you should traverse all layouts and detach them explicitely.

Accessing All Sublayouts

When the Recursive Layout is attached, you can conveniently access all layouts that will be used during layout. This works in all layout modes:

```
Enumeration getLayouts(boolean preOrder)
```

It returns an enumeration of instances of `IlvGraphLayout`. If the `preOrder` flag is `true`, the layout of the parent graph occurs before the layout of its children in the enumeration. If the `preorder` flag is `false`, the layout of the parent graph occurs after the layout of its children. For instance, in the graph of Figure 4.131 on page 276, the call `getLayouts(true)` returns the layouts for the subgraphs in this order: L1, L1.1, L1.1.1, L1.1.2, L1.2. The call `getLayouts(false)` returns the layouts for the subgraphs in this order: L1.1.1, L1.1.2, L1.1, L1.2, L1.

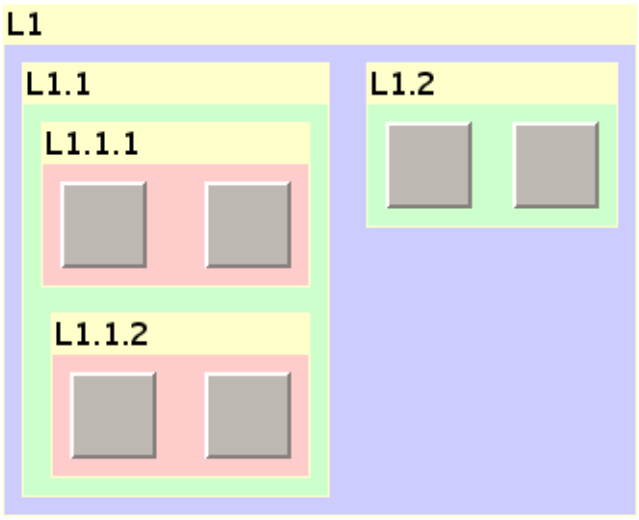


Figure 4.131 *Nesting Structure in a Graph*

> ***Note:*** *In specified provider mode, the enumeration returned by* `getLayouts` *contains the instances that are delivered by the specified provider. If the specified provider returns a different instance in each call of* `getGraphLayout(IlvGraphModel)`*, then the enumeration does not contain the instances that are later used during layout. Hence it is recommended to use layout providers that store the layout instances internally and return the same instance for the same graph model. The predefines* `IlvDefaultLayoutProvider` *and* `IlvRecursiveLayoutProvider` *store the layout instances internally.*

Convenience Mechanism of the Reference Layout Mode

The class `IlvGraphLayout` contains a convenience method. In order to perform a recursive layout recursively, you can use:

```
int performLayout(boolean force, boolean redraw, boolean traverse)
```

If the `traverse` flag is `true`, it traverses the nested graph and performs the layout on each subgraph. In fact, this is just a shortcut for the reference mode of Recursive Layout. The statement

```
flatLayout.performLayout(force, redraw, true);
```

is equivalent to creating a Recursive Layout in reference mode, that uses the `flatLayout` as reference layout:

```
IlvRecursiveLayout recursiveLayout = new IlvRecursiveLayout(flatLayout);
```

```
recursiveLayout.performLayout(force, redraw);
```

Specific Parameters

Besides some expert parameters, Recursive Layout does not provide any specific layout parameters. You can set specific layout parameters of the sublayouts individually by accessing them via `getLayout(Object)`:

```
IlvGraphLayout sublayout = recursiveLayout.getLayout(subgraph);
sublayout.set<i>Parameter</i>(<i>parameter</i>);
```

However, Recursive Layout has some convenient methods that automatically traverse the nested graph recursively and set the corresponding parameter at each sublayout of a subgraph that supports this parameter. This works in particular well in reference layout mode. In internal or specified provider mode, it takes only the current nesting structure into account. If you change the specific layout of a subgraph or the nesting structure (e.g., by adding a new subgraph) after using such a convenience method, the new layout of the new subgraph usually has a different setting, so you may need to apply the conveniance method again.

The following methods traverse the nested graph recursively and set the corresponding parameter at all sublayouts (see ***Layout Parameters and Features in IlvGraphLayout*** on page 20 and ***Chapter 5, Using Advanced Features*** for details):

- `setCoordinatesMode(int mode)`
- `void setUseDefaultParameters(boolean option)`
- `void setMinBusyTime(long time)`
- `void setInputCheckEnabled(boolean enable)`
- `void propagateLayoutOfConnectedComponentsEnabled(boolean enable)`
- `void propagateLayoutOfConnectedComponents(IlvGraphLayout layout)`
- `void propagateLinkConnectionBoxInterface(IlvLinkConnectionBoxInterface linkConnectionBoxInterface)`
- `propagateLinkClipInterface(IlvLinkClipInterface linkClipInterface)`
- `void checkAppropriateLinks()`
- `void setLinkCheckEnabled(boolean enable)`
- `void setConnectionPointCheckEnabled(boolean enable)`

For Experts: Mechanisms for Advanced Users

This section explains some mechanisms that are available for advanced users:

- How to correct the subgraphs during layout
- How to install layout event listeners efficiently

Subgraph Correction

In ILOG JViews, the position of a subgrapher (instance of `IlvGrapher`) is always calculated from the positions of the contents of the subgrapher. The position of the subgrapher is simply the position of the bounding box around its contents. This mechanism has side effects when performing layout: a subgraph will never appear fixed if its contents is laid out, because by performing a layout on the contents of the subgraph, the bounding box, hence the position, of the subgraph changes. Depending on the applications, this may be undesired.

Normally, after the subgraph is laid out, its parent graph gets laid out, which will move the entire subgraph to its final position. Therefore, in most cases, the undesired position of the subgraph is only temporary and you can ignore the entire problem. However, in a few situations, you need to be aware of the effect:

- If the subgraph is specified as fixed (for instance, by using `setFixed`) and hence should not move.
- If the parent graph is never laid out.
- If the parent graph is laid out in a layout style in an incremental mode that analyzes the positions of the nodes (for instance in Tree Layout and Hierarchical Layout).

In all these situations, it is important that the subgraph remains at the old position even after its contents was laid out. The implementation of `IlvGrapher` does not behave like this.

The solution is simply to move the subgraph back to its old position immediately after the subgraph was laid out, just before the layout of the parent graph of the subgraph is started. The Recursive Layout allows to install a subgraph correction interface that contains a `correct` method which is called exactly at this time point. You install a subgraph correction interface in the following way:

```
layout.setSubgraphCorrectionInterface(
    new IlvSubgraphCorrectionBarycenterFixed());
```

Two default implementations of `IlvSubgraphCorrectionInterface` are available:

- `IlvSubgraphCorrectionBarycenterFixed` corrects the subgraph so that after the layout of its contents the center of the subgraph remains the same. Still the size of the bounding box of the subgraph may change due to the layout of the contents.
- `IlvSubgraphCorrectionBoundsFixed` corrects the subgraph so that after the layout of its contents the bounding box of the subgraph remains the same. However, in order to achieve this, the zoom level of the subgraph is changed.

These implementation of the subgraph correction interface do not correct the top-level graph, but only subgraphs that are nested inside another grapher. The instances can be shared between different instances of `IlvRecursiveLayout`.

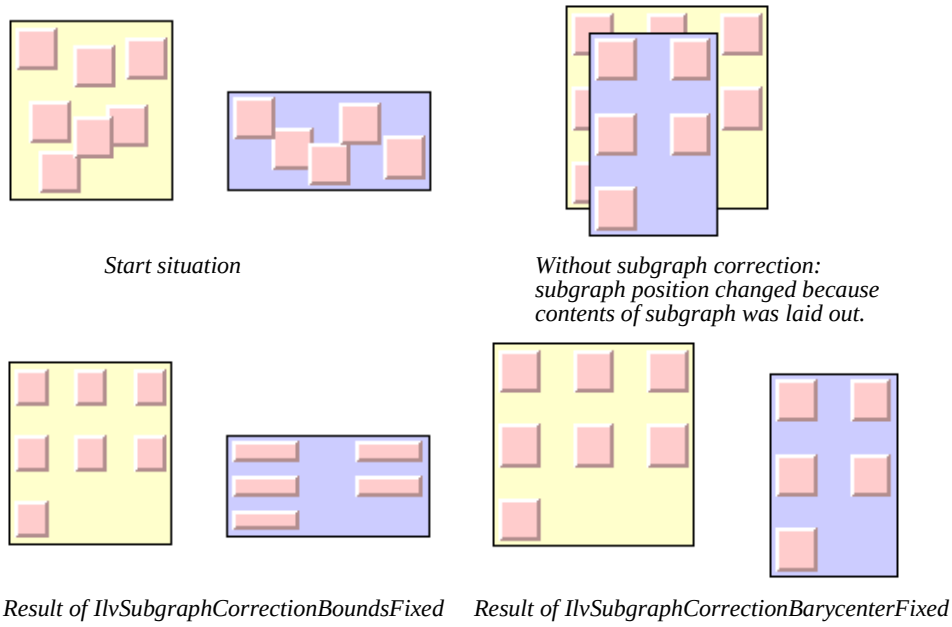


Figure 4.132 *Effect of Subgraph Correction*

Layout Listeners

Layout event listeners are an advanced feature described in general in *Using Event Listeners* on page 294. You should come back to this section after you have understood the concept of layout listeners. We mention here some specific details of the Recursive Layout related to layout listeners.

The application can listen to the layout events that the Recursive Layout sends, or it can listen to the layout events that each sublayout sends individually. For instance, a progress bar that displays the progress of the entire nested layout should listen to the layout events that the Recursive Layout fires itself, while an application that wants to detect when a specific sublayout of a subgraph is started or stopped should listen to the layout events of this particular sublayout.

In order to install a layout event listener at the Recursive Layout, call usually:

```
recursiveLayout.addGraphLayoutEventListener(listener);
```

In order to install a layout listener that receives the layout events of all sublayouts of the Recursive Layout, you can call:

```
recursiveLayout.addSubLayoutEventListener(listener);
```

Notice that in this case, the listener is installed at the Recursive Layout instance (not at the sublayout instances) but receives the events from the sublayouts (not from the Recursive Layout). An internal mechanism makes sure that the events are forwarded to the listener.

Alternatively, you could traverse the nesting structure and install the same listener at all layouts of subgraphs. However, this would have two disadvantages: it requires more memory, and you need to reinstall or update the listener whenever you change the layout of a subgraph or the nesting structure by adding or removing subgraphs. When you use `addSubLayoutEventListener`, the update of the listener is not necessary in this case.

For Experts: More on Layout Providers

We illustrated above how to use the Recursive Layout when using a specified layout provider. The library provides a default implementation of the interface `IlvLayoutProvider`, named `IlvDefaultLayoutProvider`. In many cases, it is simpler either to use this class as is, or to subclass it, rather than directly implementing this interface.

The class `IlvDefaultLayoutProvider` allows you to set the layout instance to be used for each graph (called the *preferred layout*) with the method:

```
setPreferredLayout(IlvGraphModel graphModel, IlvGraphLayout
layout, boolean detachPrevious)
```

The layout instance specified as the preferred layout is stored in a property of the graph. The current preferred layout is returned by the method:

```
getPreferredLayout(IlvGraphModel graphModel)
```

The method returns `null` if no layout has been specified for this graph.

When the method `getGraphLayout` is called on the default provider, the previously specified preferred layout is returned, if any. Otherwise, a new layout instance is allocated by a call to the method

```
createGraphLayout(IlvGraphModel graphModel)
```

This newly created layout is recorded as the preferred layout of this graph, which is attached to the layout instance.

The default implementation of the method `createGraphLayout` copies the layout instance that is the preferred layout of the nearest parent graph of the given graph for which a preferred layout has been specified. Therefore, if a preferred layout `L` is specified for a graph `G` and no preferred layout is set for its subgraphs, then the graph `G` and all its subgraphs are laid out using the same layout algorithm `L` (copies of it are used for the subgraphs).

> ***Note:*** *You must call the method* `detachLayouts(IlvGraphModel graphModel, boolean traverse)` *when you no longer need the layout provider instance; otherwise, some objects may not be garbage collected.*

The settings of the preferred layout made using the class `IlvDefaultLayoutProvider` can be saved in IVL files. For details, see *Saving Layout Parameters and Preferred Layouts* on page 315.

Code Sample

The following code sample illustrates the use of the class `IlvDefaultLayoutProvider`:

```
...
IlvGrapher grapherA = new IlvGrapher();
IlvGrapher grapherB = new IlvGrapher();

// Fill the graphers with nodes and links;
// grapherB is added as a subgraph of grapherA
grapherA.addNode(grapherB, false);

// Create a grapher adapter for the topmost graph
IlvGrapherAdapter adapterA = new IlvGrapherAdapter(grapherA);

// Get a grapher adapter for the subgraph
IlvGraphModel adapterB = adapterA.getGraphModel(grapherB);

// Create the layout provider
IlvDefaultLayoutProvider provider = new IlvDefaultLayoutProvider();

// Specify the preferred layouts for each grapher
// (this automatically attaches the layouts)
provider.setPreferredLayout(adapterA, new IlvTreeLayout());
provider.setPreferredLayout(adapterB, new IlvGridLayout());

// Create a recursive layout in specified provider mode
IlvRecursiveLayout layout = new IlvRecursiveLayout(provider);

// Perform the layout
try {
        IlvRecursiveLayoutReport layoutReport =
              (IlvRecursiveLayoutReport)layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
...
// detach the layouts when the provider is no longer needed
provider.detachLayouts(adapterA, true);
// dispose the topmost adapter when no longer needed
adapterA.dispose();
...
```

Multiple Layout

In this section, you will learn about the *Multiple Layout* class from the ILOG JViews Graph Layout module (class `IlvMultipleLayout` from the package `ilog.views.graphlayout.multiple`).

Application Domain

The Multiple Layout class is not a layout algorithm but rather a facility to compose multiple layout algorithms and treat them as one algorithm object. This is necessary in particular when dealing with the recursive layout of nested submanagers (see *Recursive Layout* on page 265 and *Laying Out Nested Graphs* on page 306) because performing the layouts recursively one after the other has a different effect than combining the layouts into one algorithm object and applying this object all at once. Multiple Layout should also be used to combine a normal layout with a Link Layout that routes intergraph links. This is illustrated in the following sample.

Code Sample

You can, for instance, combine a Tree Layout, a Link Layout, and an Annealing Label Layout into one object of type `IlvGraphLayout` in the following way:

```
 ...
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.multiple.*;
import ilog.views.graphlayout.tree.*;
import ilog.views.graphlayout.link.*;
import ilog.views.graphlayout.labellayout.annealing.*;

IlvTreeLayout treeLayout = new IlvTreeLayout();
IlvLinkLayout linkLayout = new IlvLinkLayout();
IlvAnnealingLabelLayout labelLayout = new IlvAnnealingLabelLayout();
IlvMultipleLayout multipleLayout =
        new IlvMultipleLayout(treeLayout, linkLayout, labelLayout);

IlvGrapher grapher = ...
layout.attach(grapher);

... /* Fill in code to set the layout parameters of treeLayout,
     * linkLayout and labelLayout.
     */
linkLayout.setIntergraphLinksMode(true);
...
```

By constructing a Multiple Layout instance in this way, the Tree Layout, Link Layout, and Label Layout become *sublayouts* of the Multiple Layout instance. Attaching the Multiple Layout will automatically attach its sublayouts.

The Multiple Layout has two slots for graph layouts and one slot for the label layout. Not all slots need to be used. You can pass `null` as the sublayout for unused slots. If you need more slots, you can compose a Multiple Layout that contains another Multiple Layout as a sublayout.

To perform the composed layout you use either a:

- *Simple Layout*
- *Recursive Layout*

Simple Layout

You can perform the composed layout on a flat grapher (one which contains no submanagers) in the following way:

```
try {
        IlvMultipleLayoutReport layoutReport =
              (IlvMultipleLayoutReport)multipleLayout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

The statement with `multipleLayout.performLayout()` in this case has the same effect as the sequence of the three statements:

```
treeLayout.performLayout();
linkLayout.performLayout();
labelLayout.performLayout();
```

Recursive Layout

If you perform the Multiple Layout on a grapher that contains submanagers, there is a difference in the order of the layout (see *Order of Layouts in Recursive Layouts* on page 307. You apply a recursive layout on the grapher and its submanagers in the following way:

```
IlvRecursiveLayout recursiveLayout = new IlvRecursiveLayout(multipleLayout);
try {
        IlvGraphLayoutReport layoutReport = recursiveLayout.performLayout();
        ...
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

or alternatively, in the following way (both ways are equivalent):

```
try {
```

```
        IlvMultipleLayoutReport layoutReport =
                (IlvMultipleLayoutReport)multipleLayout.performLayout(
                                                        true, true, true);
        ...
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Assume the attached grapher A contains a subgrapher B. The combined Multiple Layout applies its sublayouts in postorder, as follows:

1. Tree Layout on B
2. Link Layout on B
3. Label Layout on B
4. Tree Layout on A
5. Link Layout on A
6. Label Layout on A

This means that all layouts of subgrapher B have finished before the layout of grapher A starts. This is the correct order for a recursive layout.

If you do not combine the three component layouts into a Multiple Layout, you can only apply them sequentially:

```
treeLayout.performLayout(true, true, true);
linkLayout.performLayout(true, true, true);
labelLayout.performLayout(true, true, true);
```

The effect of these three statements is slightly different than the effect of the Multiple Layout. The layouts are now applied in the following order:

1. Tree Layout on B
2. Tree Layout on A
3. Link Layout on B
4. Link Layout on A
5. Label Layout on B
6. Label Layout on A

This order is not usually suitable for the layout of nested graphers because the Tree Layout of grapher A is started too early. The Label Layout on grapher B in Step 5 may change the position of grapher B within grapher A, invalidating the result of the Tree Layout in Step 2. Hence, it is recommended that you combine multiple layout algorithms into one Multiple Layout object and apply this object as a whole to a nested grapher.

Features

- Allows composing two graph layout algorithms and one label layout algorithm into one layout object.
- Should be used to achieve the correct layout order when dealing with nested graphers.
- Layout features, speed, and quality depend on the features, speed, and quality of the sublayouts.

Generic Features and Parameters

Depending on the support of its sublayouts, Multiple Layout may support the following generic parameters defined in the `IlvGraphLayout` class:

- *Allowed Time*
- *Layout of Connected Components*
- *Percentage Completion Calculation*
- *Save Parameters to Named Properties*
- *Stop Immediately*

The following paragraphs describe the particular way in which these parameters are used by this subclass.

Allowed Time

A Multiple Layout instance supports this feature if all of its sublayouts support the feature. If the allowed time setting has elapsed, the Multiple Layout stops; that means it stops the currently running sublayout and skips the subsequent sublayouts that have not yet been started. If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Layout of Connected Components

The Multiple Layout instance can use the generic mechanism to lay out connected components if the sublayouts of type `IlvGraphLayout` support this feature. The sublayout of type `IlvLabelLayout` does not need special handling of connected components. (For more information about this mechanism, see *Layout of Connected Components* on page 25.)

Percentage Completion Calculation

The Multiple Layout calculates the percentage of completion. This value can be obtained from the layout report during the run of the layout. The value is, however, a very rough estimation. If the sublayouts do not support the calculation of the percentage completion, the Multiple Layout can report the percentage based only on the information that the sublayout has already finished. For instance, if there are three sublayouts, the mechanism reports 33%

after the first sublayout has finished, 66%after the second sublayout has finished, and 100% after all three sublayouts have finished. If the sublayouts support the calculation of the percentage completion, the Multiple Layout calculates a more detailed percentage. (For a detailed description of this feature, see *Percentage Completion Calculation* on page 31 and *Layout Event Listeners* on page 294.)

Save Parameters to Named Properties

The Multiple Layout instance can save its layout parameters into named properties if all its sublayouts support this feature. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Save Parameters to Named Properties* on page 34 and *Saving Layout Parameters and Preferred Layouts* on page 315.)

Stop Immediately

The Multiple Layout instance supports this feature if all its sublayouts support this feature. It stops the currently running sublayout after cleanup if the method `stopImmediately` is called and skips the subsequent sublayouts that have not yet been started. (For a description of this method in the `IlvGraphLayout` class, see *Stop Immediately* on page 35.) If the layout stops before completion, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific Parameters

Multiple Layout does not provide any specific layout parameters. However, you can set the generic and specific layout parameters of the sublayouts individually. For instance, you can construct a Multiple Layout instance from two graph layouts. Even though the Multiple Layout does not support setting fixed nodes on itself, you can fix nodes for the sublayouts individually by applying `setFixed` to the sublayout instances if the sublayouts support this feature:

```
IlvMultipleLayout multipleLayout =
        new IlvMultipleLayout(layout1, layout2, null);
multipleLayout.attach(grapher);
if (layout1.supportsPreserveFixedNodes()) {
        layout1.setFixed(node1, true);
        ...
}
if (layout2.supportsPreserveFixedNodes()) {
        layout2.setFixed(node2, true);
        ...
}
try {
        // perform a multiple layout: node1 is fixed while layout1 runs
        // and node2 is fixed while layout2 runs
        IlvMultipleLayoutReport layoutReport =
                (IlvMultipleLayoutReport)multipleLayout.performLayout(
                                                        true, true, true);
        ...
```

```
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Accessing Sublayouts

You can obtain the sublayouts of a Multiple Layout instance by the following methods:

- `IlvGraphLayout getFirstGraphLayout()`

  which returns the graph layout that is applied first.

- `IlvGraphLayout getSecondGraphLayout()`

  which returns the graph layout that is applied second.

- `IlvLabelLayout getLabelLayout()`

  which returns the label layout that is applied last.

You can also change the sublayouts. Of course, you should not change the sublayouts while the Multiple Layout instance is attached to a graph. You should detach the graph first.

To set the sublayouts, the following methods are available:

```
void setFirstGraphLayout(IlvGraphLayout layout)
void setSecondGraphLayout(IlvGraphLayout layout)
void setLabelLayout(IlvLabelLayout layout)
```

For Experts: Attaching Graph and Labeling Models

If a graph model is attached to the Multiple Layout instance, the same graph model is automatically attached to the sublayouts of type `IlvGraphLayout`. For the sublayout of type `IlvLabelLayout`, a default labeling model is used when possible (`IlvDefaultLabelingModel`, see *The Labeling Model* on page 346). This works if the nodes, links, and labels are stored in an `IlvGrapher`.

If you implement your own labeling model (subclass of `IlvLabelingModel`), you can force the Multiple Layout to use this labeling model instead of the default labeling model. Before you attach the graph model, you call the method `setLabelingModel` in the following way:

```
multipleLayout.setLabelingModel(myLabelingModel);
multipleLayout.attach(myGraphModel);
```

You must specify the labeling model in this way if your nodes, links, and labels are not stored in an `IlvGrapher` but in your own data structures, because the default labeling model is designed to handle only an`IlvGrapher`.

If the Multiple Layout instance is detached from the graph model, all sublayouts are automatically detached as well.

Combining Multiple and Recursive Layout

Often, the Multiple Layout is used inside a Recursive Layout. For convenience, ILOG JViews provides a layout algorithm that combines both mechanisms: the Recursive Multiple Layout. This is a Recursive Layout (see *Recursive Layout* on page 265) that uses an instance of Multiple Layout for each subgraph.

In order to apply a Tree Layout, a Link Layout, and an Annealing Label Layout recursively on a nested graph, you can use:

```
IlvRecursiveMultipleLayout layout = new IlvRecursiveMultipleLayout(
                                          new IlvTreeLayout(),
                                          new IlvLinkLayout(),
                                          new IlvAnnealingLabelLayout());
```

This is in principle the same as a Recursive Layout that has a Multiple Layout as a reference layout:

```
IlvRecursiveLayout layout = new IlvRecursiveLayout(
                                    new IlvMultipleLayout(
                                           new IlvTreeLayout(),
                                           new IlvLinkLayout(),
                                           new IlvAnnealingLabelLayout()));
```

The Recursive Multiple Layout has a first and second graph layout instance per subgraph, and a label layout instance per subgraph. You access these instances by the following methods:

- `IlvGraphLayout getFirstGraphLayout(Object subgraph)`

  which returns the graph layout that is applied first to the subgraph.

- `IlvGraphLayout getSecondGraphLayout(Object subgraph)`

  which returns the graph layout that is applied secondly to the subgraph.

- `IlvLabelLayout getLabelLayout(Object subgraph)`

  which returns the label layout that is applied last to the subgraph.

If the `subgraph` parameter is `null` in these methods, the layout instances of the top-level graph are returned.

For Experts: The Reference Labeling Model

The Recursive Multiple Layout must be used when the label layout should use a specified labeling model that is not the default labeling model (`IlvDefaultLabelingModel`, see *The Labeling Model* on page 346). The Multiple Layout allows you to specify a particular labeling model by using the method `setLabelingModel`, but when you encapsulate the Multiple Layout into a Recursive Layout, this specification would need to be repeated for each layout instance of each subgraph. This would be inconvenient. However, the Recursive Multiple Layout takes care of this mechanism automatically.

If you implement your own labeling model (subclass of `IlvLabelingModel`), you must implement the method:

```
IlvLabelingModel createLabelingModel(Object subgraph)
```

This method should return a new instance of your own labeling model for the input subgraph. The Recursive Multiple Layout uses this method to generate all labeling models for all subgraph from a reference labeling model. Before attaching the Recursive Multiple Layout instance, you can set the reference labeling model in the following way:

```
recursiveMultipleLayout.setReferenceLabelingModel(myLabelingModel);
recursiveMultipleLayout.attach(myGraphModel);
```

The reference labeling model is used for the label layout of the top-level grapher. Clones of the reference labeling model obtained by `createLabelingModel` are used for the label layouts of the subgraphers.

A simple way to perform a label layout recursively is the following:

```
IlvRecursiveMultipleLayout layout =
    new IlvRecursiveMultipleLayout(labelLayout);
layout.setReferenceLabelingModel(myLabelingModel);
layout.attach(topLevelGraph);
try {
        IlvGraphLayoutReport layoutReport = layout.performLayout();
        ...
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}

layout.detach();
```

If the Recursive Multiple Layout instance is detached from the top level graph model, all sublayouts are automatically detached as well and all labeling models of subgraphs (including the reference labeling model) are disposed of.

C H A P T E R

5

Using Advanced Features

This chapter describes advanced features for using the Graph Layout module of ILOG JViews. The following topics are covered:

- *Using a Layout Report*
- *Using Event Listeners*
- *Redrawing the Grapher after Layout*
- *Using the Graph Model*
- *Laying Out a Non-JViews Grapher*
- *Laying Out Connected Components of a Disconnected Graph*
- *Laying Out Nested Graphs*
- *Saving Layout Parameters and Preferred Layouts*
- *Using the Filtering Features to Lay Out a Part of an IlvGrapher*
- *Choosing the Layout Coordinate Space*
- *Releasing Resources Used During the Layout*
- *Defining a New Type of Layout*
- *Questions and Answers about Using the Layout Algorithms*

Using a Layout Report

Layout reports are objects used to store information about the particular behavior of a layout algorithm. After the layout is completed, this information is available to be read from the layout report.

Layout Report Classes

Each layout class instantiates a particular class of `ilog.views.graphlayout.IlvGraphLayoutReport` each time the layout is performed. Table 5.1 shows the layout classes and their corresponding layout reports.

Table 5.1 *Layout Report Classes*

Layout Class	Layout Report Class
`IlvTopologicalMeshLayout`	`IlvTopologicalMeshLayoutReport`
`IlvSpringEmbedderLayout`	`IlvGraphLayoutReport`
`IlvUniformLengthEdgesLayout`	`IlvUniformLengthEdgesLayoutReport`
`IlvTreeLayout`	`IlvGraphLayoutReport`
`IlvHierarchicalLayout`	`IlvGraphLayoutReport`
`IlvLinkLayout`	`IlvGraphLayoutReport`
`IlvRandomLayout`	`IlvGraphLayoutReport`
`IlvBusLayout`	`IlvGraphLayoutReport`
`IlvCircularLayout`	`IlvGraphLayoutReport`
`IlvGridLayout`	`IlvGraphLayoutReport`
`IlvMultipleLayout`	`IlvMultipleLayoutReport`
`IlvRecursiveLayout`	`IlvRecursiveLayoutReport`

Creating a Layout Report

All layout classes inherit the `performLayout` method from the `IlvGraphLayout` class. This method calls `createLayoutReport` to obtain a new instance of the layout report. This instance is returned when `performLayout` returns. The default implementation in the base layout class creates an instance of `IlvGraphLayoutReport`. Some subclasses override this method to return an appropriate subclass. Other classes, such as `IlvRandomLayout`, do not

need specific information to be stored in the layout report and do not override `createLayoutReport`. In this case, the base class `IlvGraphLayoutReport` is used.

When using the layout classes provided with ILOG JViews, you do not need to instantiate the layout report yourself. This is done automatically.

Reading a Layout Report

To read a layout report, all you need to do is store the layout report instance returned by the `performLayout` method and read the information, as shown in the following example for the Spring Embedder Layout:

```
 ...
try {
        IlvGraphLayoutReport layoutReport = layout.performLayout();
        if (layoutReport.getCode() ==
                                    IlvGraphLayoutReport.LAYOUT_DONE)
                System.out.println("Layout done.");
        else
                System.out.println("Layout not done, code = " +
                                   layoutReport.getCode());
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Information Stored in a Layout Report

The base class `IlvGraphLayoutReport` stores the following information:

- Code

  This field contains information about special, predefined cases that may have occurred during the layout. The possible values are the following:

  - `LAYOUT_DONE` appears if the layout was performed successfully.
  - `STOPPED_AND_VALID` appears if the layout was performed but was stopped before completion, either because the layout time elapsed or because the method `stopImmediately` was called. The positions of nodes and links are valid at the stopping point because the layout algorithm uses an iterative mechanism.
  - `STOPPED_AND_INVALID` appears if a (noniterative) layout was performed but was stopped before completion, either because the layout time elapsed or because the method `stopImmediately` was called. The positions of nodes and links are not valid at the stopping point. Often, they have not yet been changed at all.
  - `NOT_NEEDED` appears if the layout was not performed because no changes occurred in the grapher and parameters since the last time the layout was performed successfully.

- `EMPTY_GRAPHER` appears if the grapher is empty.

To read the code, use the method:

```
int getCode()
```

- Layout Time

  This field contains the total duration of the layout algorithm at the end of the layout. To read the time (in milliseconds), use the method:

  ```
  long getLayoutTime()
  ```

- Percentage of Completion

  This field contains an estimation of the percentage of the layout that has been completed. This can be used if the layout algorithm supports the generic percentage completion calculation feature. (See *Percentage Completion Calculation* on page 31.) It is typically used inside layout event listeners that are described in the following section. To access the percentage, use the method:

  ```
  int getPercentageComplete()
  ```

- Additional information

  Additional information for particular layout algorithms is stored by the subclasses of `IlvGraphLayoutReport`. For details, see the *Reference Manual* for these classes:

  - `IlvTopologicalMeshLayoutReport`
  - `IlvUniformLengthEdgesLayoutReport`

Using Event Listeners

All layout classes support two kinds of events: layout events and parameter events.

Layout Event Listeners

The layout event listeners mechanism provides a way to inform the end user of what is happening during the layout. At times, a layout algorithm may take a long time to execute, especially when dealing with large graphs. In addition, an algorithm may not converge in some cases. No matter what the situation, the end user should be informed of the events that occur during the layout. This can be done by implementing a simple progress bar or by displaying appropriate information, such as the percentage of completion after each iteration or step.

The layout event listener is defined by the `GraphLayoutEventListener` interface. To receive the layout events delivered during the layout, a class must implement the

GraphLayoutEventListener interface and must register itself using the addGraphLayoutEventListener method of the IlvGraphLayout class.

When you implement the GraphLayoutEventListener interface, you must implement the layoutStepPerformed method. The layout algorithm will call this method on all the registered layout event listeners, passing the layout report as an argument. In this way, you can read information about the current state of the layout report. (For example, you can read this information after each iteration or step of the layout algorithm).

The following example shows how to implement a layout event listener:

```
class LayoutIterationListener
  implements GraphLayoutEventListener
{
  public void layoutStepPerformed(GraphLayoutEvent event)
  {
    IlvGraphLayoutReport layoutReport = event.getLayoutReport();
    System.out.println("percentage of completion: " +
                       layoutReport.getPercentageComplete());
  }
}
```

Then, register the listener on the layout instance as follows:

```
layout.addGraphLayoutEventListener(new LayoutIterationListener());
```

Parameter Event Listeners

The layout parameter event listeners mechanism provides a way to inform the end user that any layout parameter has changed. This is useful when the layout parameter values are displayed in a dialog box that needs to be updated to indicate parameter changes.

The parameter event listener is defined by the GraphLayoutParameterEventListener interface. To receive the layout parameter events, a class must implement the GraphLayoutParameterEventListener interface and must register itself using the addGraphLayoutParameterEventListener method of the IlvGraphLayout class.

When you implement the GraphLayoutParameterEventListener interface, you must implement the parametersUpToDate method. The layout class will call this method on all the registered layout parameter event listeners. The layout parameter event contains a flag accessible by the isParametersUpToDate() method:

- It returns true if the event occurs at the end of a run of the layout when the layout is considered up-to-date with respect to the layout parameters.
- It returns false if the event occurs when any layout parameter has changed.

The following example shows how to implement a layout parameter event listener.

```
class LayoutParameterListener
  implements GraphLayoutParameterEventListener
```

```
{
  public void parametersUpToDate(GraphLayoutParameterEvent event)
  {
    if (!event.isParametersUpToDate())
      System.out.println("Any layout parameter has changed.”);
  }
}
```

Then, register the listener with the layout instance as follows:

```
layout.addGraphLayoutParameterEventListener(new LayoutParameterListener());
```

Redrawing the Grapher after Layout

When a layout algorithm is executed, it moves the nodes and/or reshapes the links of the graph. If the graph is displayed on a screen, its display must be updated to reflect the changes made by the layout. Several scenarios for updating the display are possible, one or another being preferable depending on the particular application. ILOG JViews provides complete flexibility concerning the redraw operation. When laying out an ILOG JViews grapher (`IlvGrapher`), the following scenarios are possible:

- Automatic and selective redraw

  If you just want the grapher to be automatically redrawn after the layout, simply call the following method with the value `true` for the `redraw` argument:

  ```
  IlvGraphLayoutReport performLayout(boolean force, boolean redraw)
  ```

  When you do this, an `initReDraws/reDrawViews` session is initiated automatically. When the nodes and the links are moved or reshaped, the value `true` is passed for the `redraw` argument of the appropriate methods of `IlvGrapher`. At the end of the layout, the `initReDraws/reDrawViews` session is ended. This produces a selective redraw of the invalid regions of the views where the graph is displayed.

- Nonautomatic and complete redraw

  If all the nodes are moved by the layout, it may be more efficient to redraw the entire graph at the end of the layout instead of using the mechanism of the invalid regions provided by `IlvGrapher`. In this case, you can use the following code:

  ```
  try {
    layout.performLayout(false, false); // argument redraw at false
  } catch (IlvGraphLayoutException e) {
    e.printStackTrace();
  } finally {
    // redraw in the final clause to ensure that the redraw
    // is performed even if an exception occurs
    grapher.reDraw();
  }
  ```

This completely redraws the grapher in all its visible views. Alternatively, you can call the method `repaint` on some of its views.

- Delayed redraw

  After the layout, you may want to perform other changes in the grapher before redrawing. You can start the `initReDraws/reDrawViews` session on your own to control the point in time when the redraw is performed. You can use the following code:

```
grapher.initReDraws();
try {
  layout.performLayout(false, true);
} catch (IlvGraphLayoutException e) {
  e.printStackTrace();
} finally {
  grapher.moveObject(..., false); // some other changes in the grapher
  grapher.reDrawViews();
}
```

- No redraw at all

  Sometimes, the layout may need to be performed without any display of the grapher. For instance, this can be done to automatically produce `.ivl` files containing the result of the layout for future use. To avoid any redraw during the layout, just call the method `performLayout` with the value `false` for the `redraw` argument.

> ***Note:*** *During animated layout (for layouts supporting this option), the grapher needs to be redrawn after each step to produce the animation effect. Therefore, you need to pass the value* `true` *for the* `redraw` *argument of the method* `performLayout(boolean, boolean)`. *In this case, the* `initReDraw/reDrawViews` *session is automatically used for each animation step. Users should not add their own* `initReDraw/reDrawViews` *session because this would prevent the graph from being redrawn during the animation.*

Using the Graph Model

The `ilog.views.graphlayout.IlvGraphModel` class defines a suitable, generic API for graphs that have to be laid out with ILOG JViews graph layout algorithms.

All the layout algorithms provided in ILOG JViews are designed to lay out a graph model. This allows applications to benefit from the graph layout algorithms whether or not they use the ILOG JViews grapher (`ilog.views.IlvGrapher`). However, to make things very simple for the common case of applications that manipulate an `IlvGrapher`, it is not mandatory to work directly with the graph model except for some advanced features such as filtering (see *Using the Filtering Features to Lay Out a Part of an IlvGrapher* on page 321).

The Graph Model Concept

With a graph model, you can use already-built graphs, nodes, and links that have been developed without ILOG JViews and apply the layout algorithms of ILOG JViews. The graph model provides the basic, generic operations for performing the layout. A subclass must be written to adapt the graph model to specific graph, node, and link objects. This subclass plays the role of an "adapter" or bridge between the application objects and the graph model. This often makes it much easier to add graph features to existing applications.

Figure 5.1 shows the relationship between the graph model and graph layout algorithms, ILOG JViews graphers, non-JViews graphers, and manager views.

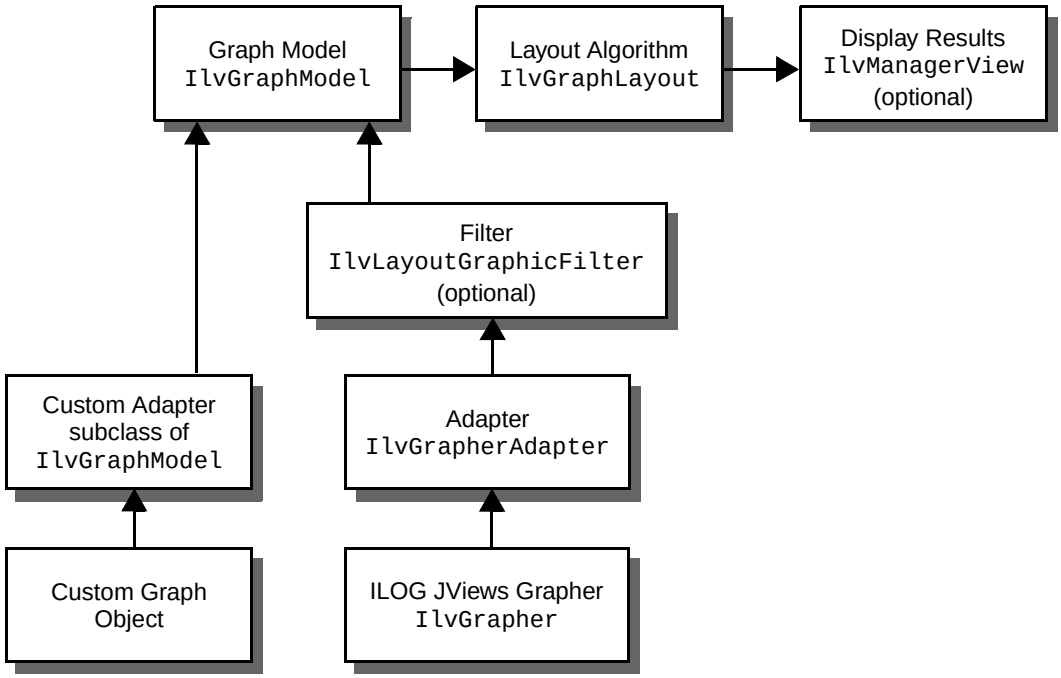


Figure 5.1 *Graph Model in the ILOG JViews Graph Layout Module*

You can see from this diagram that instead of using a concrete graph class such as `IlvGrapher` directly, the layout algorithms interact with the graph via the graph model. This is the key for achieving a truly generic graph layout framework. Note that the use of an `IlvManagerView` to display the result of the layout is not mandatory.

The IlvGraphModel Class

The `IlvGraphModel` class is an abstract Java class. Because it does not provide a concrete implementation of a graph data structure, a complete implementation must be provided by "adapter" classes. The adapters extend the `IlvGraphModel` class and must use an underlying graph data structure. A special adapter class called `IlvGrapherAdapter` is provided so that an `IlvGrapher` can be used as the underlying graph data structure.

> ***Note:*** *If an application uses the* `IlvGrapher` *class, the grapher can be attached directly to the layout instance without explicitly using a graph model. (See the* `IlvGraphLayout.attach(IlvGrapher)` *method.) In this case, the appropriate adapter (*`IlvGrapherAdapter`*) will be created internally. This adapter can be retrieved using the* `IlvGraphLayout.getGraphModel()` *method, which will return an instance of* `IlvGrapherAdapter`*.*

Most of the methods defined in the `IlvGraphModel` class have a name and definition very similar to the corresponding methods of the `IlvGrapher` class. The main difference is that the arguments of the `IlvGraphModel` methods are `java.lang.Object` instead of `IlvGraphic` or `IlvLinkImage`. The methods can be divided into several categories that provide information on the structure of the graph, the geometry of the graph, modification of the graph geometry, and notification of changes in the graph.

Information on the Structure of the Graph

The following methods of the `IlvGraphModel` class allow the layout algorithms to retrieve information on the structure of the graph:

```
Enumeration getNodesAndLinks()

Enumeration getNodes()

int getNodesCount()

Enumeration getLinks()

int getLinksCount()

boolean isNode(Object obj)

boolean isLink(Object obj)

Enumeration getLinks(Object node)

int getLinksCount(Object node)

Enumeration getLinksFrom(Object node)

int getLinksFromCount(Object node)

Enumeration getLinksTo(Object node)
```

```
int getLinksToCount(Object node)
Enumeration getNeighbors(Object node)
int getNodeDegree(Object node)
Object getFrom(Object link)
Object getTo(Object link)
Object getOpposite(Object link, Object node)
boolean isLinkBetween(Object node1, Object node2)
```

The following methods are provided for use with nested graphs (see also *Laying Out Nested Graphs* on page 306):

```
IlvGraphModel getParentModel()
IlvGraphModel getRootModel()
IlvGraphModel getGraphModel(Object subgraph)
IlvGraphModel createGraphModel(Object subgraph)
Enumeration getSubgraphs()
int getSubgraphsCount()
boolean isSubgraph(Object obj)
Enumeration getInterGraphLinks()
int getInterGraphLinksCount()
boolean isInterGraphLink(Object obj)
```

Information on the Geometry of the Graph

The following methods of the `IlvGraphModel` class allow the layout algorithms to retrieve information on the geometry of the graph:

```
IlvRect boundingBox(Object nodeOrLink)
IlvRect boundingBox()
IlvPoint[] getLinkPoints(Object link)
IlvPoint getLinkPointAt(Object link, int index)
int getLinkPointsCount(Object link)
float getLinkWidth(Object link)
```

The `boundingBox` method is called by a layout algorithm whenever it needs to get the position and the dimensions of a node or a link. The other methods are used mainly by link layout algorithms.

Modification of the Geometry of the Graph

The following methods of the `IlvGraphModel` class allow a layout algorithm to modify the geometry of the graph:

```
void moveNode(Object node, float x, float y, boolean redraw)

void reshapeLink(Object link, IlvPoint fromPoint,
                        IlvPoint[] points,
                        int startIndex, int length,
                          IlvPoint toPoint, boolean redraw)

void move(float x, float y, boolean redraw)
```

Layout algorithms that compute new coordinates for the nodes use the `moveNode` method. Link layout algorithms that compute new shapes for the links call one of the `reshapeLink` methods.

Notification of Changes

The following methods of the `IlvGraphModel` class allow a layout algorithm to be notified of changes in the graph:

```
void addGraphModelListener(GraphModelListener listener)

void removeGraphModelListener(GraphModelListener listener)

void fireGraphModelEvent(GraphModelEvent event)

void fireGraphModelEvent(Object nodeOrLink, int type, boolean
adjusting)

void adjustmentEnd()
```

A “change” in the graph can be a structure change (that is, a node or a link was added or removed) or a geometry change (that is, a node or a link was moved or reshaped). The graph model event listener mechanism provides a means to keep the layout algorithms informed of these changes. When the layout algorithm is restarted on the same graph, it is able to detect whether the graph has changed since the last time the layout was successfully performed. If necessary, the layout can be performed again. If there is no change in the graph, the layout algorithm can avoid unnecessary work by not performing the layout. To know whether the previous layout is still valid or it must be redone, the layout algorithms call the following method of the model:

```
boolean isLayoutNeeded()
```

The graph model event listener is defined by the `GraphModelListener` interface. To receive the graph model events (that is, instances of the `GraphModelEvent` class), a class

must implement the `GraphModelListener` interface and must register itself using the `addGraphModelListener` method of the `IlvGraphModel` class.

> ***Note:*** *The creation of the graph model event listener is handled transparently by the* `IlvGraphLayout` *class. Therefore, there is usually no need to manipulate this listener directly.*

Storing and Retrieving Data Objects ("Properties")

The following methods of the `IlvGraphModel` class allow a layout algorithm to store data objects for each node, link, or graph:

```
void setProperty(Object nodeOrLink, String key, Object value)

Object getProperty(Object nodeOrLink, String key)

void setProperty(String key, Object value)

Object getProperty(String key)
```

The layout algorithm may need to associate a set of properties with the nodes and links of the graph or with the graph itself. Properties are a set of `key-value` pairs, where the `key` is a `String` object and the `value` can be any kind of information value.

> ***Note:*** *Creating a property and associating it with a node, a link, or a graph is handled transparently by the layout algorithm whenever it is necessary. Therefore, there is usually no need to manipulate the properties directly. However, if needed, you can do this in your own subclass of* `IlvGraphLayout`.

Using the IlvGrapherAdapter

The `IlvGrapherAdapter` class is a concrete subclass of `IlvGraphModel` that allows an `IlvGrapher` to be laid out using the layout algorithms provided in ILOG JViews. It provides an implementation for all the abstract methods of `IlvGraphModel`. It also provides an overridden implementation of some nonabstract methods of `IlvGraphModel` to improve efficiency by taking advantage of the characteristics of the `IlvGrapher`.

If an application uses the `IlvGrapher` class, the grapher can be attached directly to the layout instance without explicitly using the adapter. (See the method `IlvGraphLayout.attach(IlvGrapher)`.) In this case, an `IlvGrapherAdapter` is created internally by the layout class. The adapter can be retrieved using the method `IlvGraphLayout.getGraphModel()`, which will return an instance of `IlvGrapherAdapter`.

Notice that such an internally created adapter is not allowed to be attached to any other layout instance, nor to be used in any way once the method `IlvGraphLayout.detach()` has been called on the layout instance.

In case you need to be able to do any of the above operations, directly create the instance of `IlvGrapherAdapter` and attach it using `IlvGraphLayout.attach(IlvGraphModel)`.

To know whether a given `IlvGraphModel` instance has been created using `IlvGraphLayout.attach(IlvGrapher)`, you can use the method `IlvGraphModel.getOriginatingLayout()`. This method returns a non-`null` object if the model has not been created using `IlvGraphLayout.attach(IlvGrapher)`.

Additionally, the `IlvGrapherAdapter` class provides a way to filter the `IlvGrapher`. By using the filtering mechanism, you specify a particular set of nodes and links that have to be taken into account by the layout algorithm. (See *Using the Filtering Features to Lay Out a Part of an IlvGrapher* on page 321.)

The `IlvGrapherAdapter` class also allows you to specify the `IlvTransformer` that has to be used for computing the geometry of the graph. (See *Choosing the Layout Coordinate Space* on page 323.)

> ***Note:*** *For details on how to write your own adapter, see Laying Out a Non-JViews Grapher on page 303.*

Laying Out a Non-JViews Grapher

> ***Note:*** *To understand this section better, read Using the Graph Model on page 297 first.*

It is sometimes necessary to add graph layout features to an existing application. If the application already uses the ILOG JViews grapher (`ilog.views.IlvGrapher`) to manipulate and display graphs, using the graph layout algorithms provided in ILOG JViews is a straightforward process. No adapter has to be written.

However, the case may arise where an application uses its own classes for nodes, links, and graphs, and where, for some reason, you do not want to replace these classes with ILOG JViews classes. To enable the graph layout algorithms to work with these graph objects, a custom adapter (that is, a subclass of `IlvGraphModel`) must be written.

The adapter must implement all the abstract methods of the `IlvGraphModel` class. The nonabstract methods of this class have a default implementation that is really functional. However, they may not be optimal because they do not take advantage of the characteristics of the underlying graph implementation. For better performance, the following nonabstract methods can be overridden in the adapter class:

```
int getNodesCount()

int getLinksCount()

int getLinksCount(Object node)
```

```
int getLinksFromCount(Object node)
int getLinksToCount(Object node)
int getLinkPointAt(Object link, int index)
int getSubgraphsCount()
int getInterGraphLinksCount()
```

The efficiency of the layout algorithm depends directly on the efficiency of the implementation of the adapter class and the underlying graph data structure.

Laying Out Connected Components of a Disconnected Graph

ILOG JViews provides special support for the layout of a disconnected graph. If a graph is composed of several connected components or contains isolated nodes (nodes without any links), it can be desirable to apply the layout algorithm separately on each connected component and then to position the connected components using a specialized layout algorithm (usually, `IlvGridLayout`). Figure 5.2 shows an example of a graph containing four connected components. Simply by enabling the layout of the connected components on the regular layout instance (here, `IlvTopologicalMeshLayout`), the connected components are automatically identified and laid out individually. Finally, the four connected components are positioned using a highly customizable placement algorithm (`IlvGridLayout`).

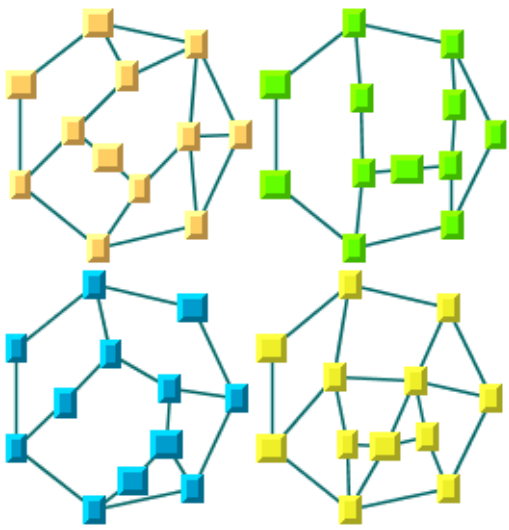


Figure 5.2 *Automatic Layout of Connected Components on a Disconnected Graph*

To indicate whether a subclass of `IlvGraphLayout` supports this feature, the following method is provided in the class `IlvGraphLayout`:

```
boolean supportsLayoutOfConnectedComponents()
```

The default implementation returns `false`. A subclass can override this method in order to return `true` to indicate that this feature is supported.

ILOG JViews allows you to enable the layout of the connected components using the method:

```
void setLayoutOfConnectedComponentsEnabled(boolean enable)
```

To obtain the current setting:

```
boolean isLayoutOfConnectedComponentsEnabled()
```

The default value is the value returned by the following method:

```
boolean isLayoutOfConnectedComponentsEnabledByDefault()
```

The default implementation of this method in `IlvGraphLayout` returns `false`. For some of the layout classes, it is appropriate that this feature is enabled by default. Therefore, `IlvSpringEmbedder` and `IlvUniformLengthEdgesLayout` override this method to return `true`.

If enabled on a layout class that supports this feature, the method `performLayout` of the class `IlvGraphLayout` cuts the attached graph model into connected components and lays out each connected component separately.

How does the layout of connected components feature work when this mechanism is enabled in the layout classes that support this feature? Instead of directly calling the method `layout` to perform the layout on the entire graph, the method `performLayout` first cuts the graph into connected components. Then, each connected component is laid out separately by a call of the method `layout`. To do this, the attached graph is temporarily changed into internally generated graphs corresponding to each of the connected components of the original graph. Finally, the layout instance returned by the method:

```
IlvGraphLayout getLayoutOfConnectedComponents()
```

is used to position the connected components. To specify the layout instance that places the connected components, use the following method:

```
void setLayoutOfConnectedComponents(IlvGraphLayout layout)
```

If no layout instance is specified using this method, the method `getLayoutOfConnectedComponents` returns an instance of `IlvGridLayout`. Its layout

region parameter is set by default to the rectangle (`0`, `0`, `800`, `800`). Its "layout mode" parameter is set to `IlvGridLayout.TILE_TO_ROWS`.

> ***Note:*** *The Tree, Hierarchical, and Circular layouts contain built-in support for disconnected graphs. For the Tree and Hierarchical layouts, the result can be different from the result of the generic mechanism (the layout of connected components feature) provided by the base class* `IlvGraphLayout`*. Depending your particular needs, you can use either the generic mechanism or the built-in support.*

Laying Out Nested Graphs

The `IlvGrapher` class provided by the ILOG JViews Graphic Framework allows nested graphs to be represented (for details see the chapter *Nested Managers and Nested Graphers* in the *ILOG JViews Graphics Framework User's Manual*). An example of a nested graph is given in Figure 5.3.

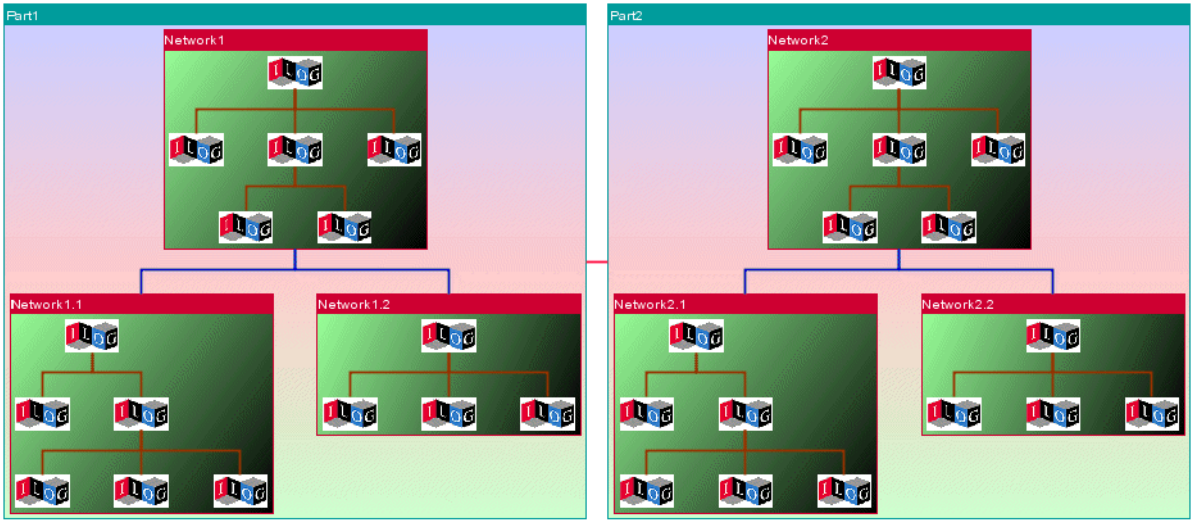


Figure 5.3 *Example of a Recursive Layout of a Nested Graph*

In this example, there is a topmost graph which contains only two nodes, named `Part1` and `Part2`, connected by a link. Each of these two nodes is another graph, composed of three nodes connected by links. Finally, each of these nodes is another graph, composed of some icons connected by links.

Such nested graphs are often used to represent the hierarchical decomposition of a network into subnetworks, more generally to mix connectivity information for several levels of detail in the same drawing.

In Figure 5.3, a layout algorithm (the Tree Layout) has been used for each of the graphs.

Order of Layouts in Recursive Layouts

Assume the grapher `A` contains a subgrapher `B`, and the subgrapher `B` contains two subgraphers `C` and `D`. The recursive layout needs to be applied in postorder, as follows:

1. Layout on `C`
2. Layout on `D`
3. Layout on `B`
4. Layout on `A`

This means that the layout of a graph is performed once all the layouts of its subgraphs have already been done. In our example, all layouts of subgrapher `B` have finished before the layout of grapher `A` starts. This is the correct order for a recursive layout. This order ensures that the layout of a subgraph does not invalidate the layout of its parent graphs.

The correct order for a recursive layout is implemented by the different methods provided by the library, presented in the following sections:

- *Applying the Same Layout Recursively*
- *Mixing Different Layouts in the Same Nested Graph*
- *Nesting Several Layouts for Each Graph in the Nested Graph*

Applying the Same Layout Recursively

This section covers the layout of a nested graph in the case where both of the following conditions hold:

- The same layout algorithm needs to be applied on the topmost graph and all its subgraphs.
- The settings of the layout algorithm (that is, the layout parameters) need to be the same for the topmost graph as for all the subgraphs.

This situation is illustrated by the example in Figure 5.3 on page 306, where a Tree Layout is applied on the topmost graph as well as on all its subgraphs. Moreover, the settings of the Tree Layout algorithm are the same for all the graphs: the application does not need, for instance, a different “flow direction” in the subgraphs than in the topmost graph.

Obtaining such recursive layouts is very easy. The class `IlvGraphLayout` provides a special version of the `performLayout` method:

```
performLayout(boolean force, boolean redraw, boolean traverse)
```

When the value `true` is passed for the last `boolean` argument, the layout is applied not only on the graph attached to the layout instance, but also, in a recursive way, to its subgraphs.

Internal Mechanism

How does this work internally? The mechanism is based on the principle that a given layout instance is used for only one graph and is not reused for its subgraphs. Therefore, the Tree Layout instance is automatically "cloned" using the `copy` method of the class `IlvGraphLayout`. The same principle holds for the graph models: a given graph model instance is used for only one graph and is not reused for subgraphs. The graph models for the subgraphs are created by calls to the `getGraphModel` method of the class `IlvGraphLayout`, which in turn creates the graph model using the method `createGraphModel`.

All these operations are done automatically, in a completely transparent way. All you have to do is to call the `performLayout` with `true` for the `traverse` argument.

If needed, you can get the layout instances applied on the subgraphs using the method:

```
getLayouts(boolean preOrder)
```

Code Sample

The following code sample illustrates how to apply a single layout algorithm to a nested graph:

```
...
IlvGrapher grapherA = new IlvGrapher();
IlvGrapher grapherB = new IlvGrapher();

// Fill the graphers with nodes and links
...
// grapherB is added as a subgraph of grapherA
grapherA.addNode(grapherB, false);

// Create the layout instance
IlvTreeLayout layout = new IlvTreeLayout();

// Attach the topmost grapher to the layout
layout.attach(grapherA);

// Perform the recursive layout
try {
        int code = layout.performLayout(true, true, true);

        System.out.println("Layout completed (code " +
          code + ")");
}
```

```
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
...
// Detach the grapher when layout no more needed
layout.detach();
...
```

An equivalent variant is the following:

```
...
IlvGrapher grapherA = new IlvGrapher();
IlvGrapher grapherB = new IlvGrapher();

// Fill the graphers with nodes and links
...
// grapherB is added as a subgraph of grapherA
grapherA.addNode(grapherB, false);

// Create the layout instance
IlvTreeLayout layout = new IlvTreeLayout();

// Create a grapher adapter for the topmost grapher
IlvGrapherAdapter adapter = new IlvGrapherAdapter(grapherA);

// Attach the adapter to the layout
layout.attach(adapter);

// Perform the recursive layout
try {
        // perform the layout with argument traverse = true
        int code = layout.performLayout(true, true, true);

        System.out.println("Layout completed (code " +
          code + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
...
// Detach the adapter when layout no more needed
layout.detach();
// Dispose the adapter when no more needed
adapter.dispose();
...
```

Layout Parameters in Simple Recursive Layouts

We have seen that, when applying the same layout algorithm in a recursive way, the layout instances for the subgraphs are obtained by "cloning" the layout instance attached to the topmost graph. The layout parameters of the "clone" are the same as the parameters of the topmost layout, except for parameters that are specific to a node or a link. Such parameters

are not copied when cloning the layouts and need to be set separately for each layout instance.

For instance, if a node `node1` contained in the subgraph `grapherB` of the topmost graph `grapherA` needs to be declared as fixed (see *Preserve Fixed Nodes* on page 32), the following code can be used:

```
...
IlvGrapher grapherA = new IlvGrapher();
IlvGrapher grapherB = new IlvGrapher();

// fill the graphers with nodes and links;
// grapherB is added as a subgraph of grapherA
grapherA.addNode(grapherB, false);

// Create the layout instance
IlvTreeLayout layout = new IlvTreeLayout();

// Attach the topmost grapher to the layout
layout.attach(grapherA);

// Ask the layout algorithm to not move the nodes
// specified as fixed. This settings is automatically
// copied on the sublayouts. Do not specify this global
// settings directly on the sublayout, because it gets automatically
// the same settings as the topmost layout
layout.setPreserveFixedNodes(true);

// Search the layout instance used for grapherB
IlvGraphLayout subLayout = null;
Enumeration layouts = layout.getLayouts(true);
while (layouts.hasMoreElements()) {
       subLayout = (IlvGraphLayout)layouts.nextElement();
       if (subLayout.getGraphModel().getGrapher() == grapherB)
              break;
}

// Specify node1 (contained in grapherB) as fixed
subLayout.setFixed(node1, true);

// Now perform the recursive layout. The node node1 will be considered as fixed
// by the layout applied to grapherB
...
```

> ***Note:*** *You should not try to change global settings of the layouts applied on the subgraphs (that is, settings that are not specific to a node or a link). Such settings are copied anyway from the layout instance of the topmost grapher, so such changes are erased just before the recursive layout runs.*

Mixing Different Layouts in the Same Nested Graph

This section covers the layout of a nested graph in the case where at least one of the following conditions holds:

- The layout algorithm to be applied on subgraphs is not the same as the algorithm needed for the topmost graph.
- Different layouts need to be applied to different subgraphs.
- The same layout algorithm, but using different settings, needs to be applied on different graphs.

Using the Layout Provider

The ILOG JViews Graph Layout library provides a flexible mechanism for the choice of the layout instance to be applied to each subgraph in a nested graph. This mechanism is based on the interface `IlvLayoutProvider`. The interface has a unique method:

```
getGraphLayout(IlvGraphModel graphModel)
```

This method must return the layout instance to be used for the graph model passed as the argument, or `null` if no layout is required for this graph. An instance of a class implementing this interface can be passed to the following method of the class `IlvGraphModel`:

```
performLayout(IlvGraphModel graphModel, IlvLayoutProvider
provider, boolean force, boolean redraw, boolean traverse)
```

or, equivalently, to the following static method of the class `IlvGraphLayout`:

```
PerformLayout(IlvGraphModel model, IlvLayoutProvider provider,
boolean force, boolean redraw, boolean traverse)
```

When performing the recursive layout, these methods get the layout instance to be used for each graph from the specified layout provider.

To implement the interface `IlvLayoutProvider`, you must decide how the choice of the layout instance is done. This can be based on some criteria such as the type of graph (eventually known in advance), or a choice already made by the end user and recorded, for instance, in a property of the graph. A possible implementation of the `getGraphLayout` method is the following:

```
public IlvGraphLayout getGraphLayout(IlvGraphModel graphModel)
{
  Object prop = graphModel.getProperty("layout type");
  // if none, return null (no layout needed for this graph)
  if (!(prop instanceof String))
    return null;

  IlvGraphLayout layout = null;
  String name = (String)prop;
```

```
  if (name.equals("tree"))
    layout = new IlvTreeLayout();
  else if (name.equals("flow"))
    layout = new IlvHierarchicalLayout();
  else
    throw new RuntimeException("unsupported layout choice: " + name);

  layout.attach(graphModel);

  return layout;
}
```

Of course, this is only an example among many possible implementations. The implementation may decide to store the newly allocated layout instance to avoid allocating a new one when the method is again called for the same graph.

The library provides a default implementation of this interface, named `IlvDefaultLayoutProvider`. In many cases, it is simpler either to use this class as is, or to subclass it, rather than directly implementing this interface.

The class `IlvDefaultLayoutProvider` allows you to set the layout instance to be used for each graph (called the *preferred layout*) with the method:

```
setPreferredLayout(IlvGraphModel graphModel, IlvGraphLayout
layout, boolean detachPrevious, boolean traverse)
```

The layout instance specified as the preferred layout is stored in a property of the graph. The current preferred layout is returned by the method:

```
getPreferredLayout(IlvGraphModel graphModel)
```

The method returns `null` if no layout has been specified for this graph.

When the method `getGraphLayout` is called on the default provider, the previously specified preferred layout is returned, if any. Otherwise, a new layout instance is allocated by a call to the method

```
createGraphLayout(IlvGraphModel graphModel)
```

This newly created layout is recorded as the preferred layout of this graph, which is attached to the layout instance.

The default implementation of the method `createGraphLayout` copies the layout instance that is the preferred layout of the nearest parent graph of the given graph for which a preferred layout has been specified. Therefore, if a preferred layout `L` is specified for a graph `G` and no preferred layout is set for its subgraphs, then the graph `G` and all its subgraphs are laid out using the same layout algorithm `L` (copies of it are used for the subgraphs).

> ***Note:*** *You must call the method* `detachLayouts(IlvGraphModel graphModel, boolean traverse)` *when you no longer need the layout provider instance; otherwise, some objects may not be garbage collected.*

The settings of the preferred layout made using the class `IlvDefaultLayoutProvider` can be saved in IVL files. For details, see *Saving Layout Parameters and Preferred Layouts* on page 315.

Code Sample

The following code sample illustrates the use of the class `IlvDefaultLayoutProvider`:

```
...
IlvGrapher grapherA = new IlvGrapher();
IlvGrapher grapherB = new IlvGrapher();

// Fill the graphers with nodes and links;
// grapherB is added as a subgraph of grapherA
grapherA.addNode(grapherB, false);

// Create a grapher adapter for the topmost graph
IlvGrapherAdapter adapterA = new IlvGrapherAdapter(grapherA);

// Get a grapher adapter for the subgraph
IlvGraphModel adapterB = adapterA.getGraphModel(grapherB);

// create the layout provider
IlvDefaultLayoutProvider provider = new IlvDefaultLayoutProvider();

// specify the preferred layouts for each grapher
// (this automatically attaches the layouts)
provider.setPreferredLayout(adapterA, new IlvTreeLayout());
provider.setPreferredLayout(adapterB, new IlvGridLayout());

// Perform the recursive layout
try {
        int code = IlvGraphLayout.PerformLayout(adapterA, provider, true, true,
                                                true);
        // or, equivalently:
        // int code = adapterA.performLayout(provider, true, true, true);

        System.out.println("Layout completed (code " +
          code + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
...
// detach the layouts when the provider is no longer needed
provider.detachLayouts(adapterA, true);
// dispose the topmost adapter when no longer needed
adapterA.dispose();
...
```

Layout Parameters in Recursive Layouts Using the Layout Provider

In the same way as for the simple recursive layouts (see *Layout Parameters in Simple Recursive Layouts* on page 309), you must specify parameters that are specific to nodes or links directly on the layout instance that lays out the subgraph containing the nodes or the links. The only thing slightly different is getting the layout instances for the subgraphs. However, a significant difference exists: when using a Layout Provider, you can specify different global settings (that is, settings that are not specific to a node or a link) for each layout instance.

The following code shows how to specify as fixed a node `node1` contained in the subgraph `grapherB` of the topmost graph `grapherA`:

```
...
IlvGrapher grapherA = new IlvGrapher();
IlvGrapher grapherB = new IlvGrapher();

// fill the graphers with nodes and links;
// grapherB is added as a subgraph of grapherA
grapherA.addNode(grapherB, false);

// Create the grapher adapter for the topmost graph
IlvGrapherAdapter adapterA = new IlvGrapherAdapter(grapherA);

// Get a grapher adapter for the subgraph
IlvGraphModel adapterB = adapterA.getGraphModel(grapherB);

// create the layout provider
IlvDefaultLayoutProvider provider = new IlvDefaultLayoutProvider();

// specify the preferred layouts for each grapher
// (this automatically attaches the layouts)
provider.setPreferredLayout(adapterA, new IlvTreeLayout());
provider.setPreferredLayout(adapterB, new IlvGridLayout());

...
// Get the layout instance for grapherB from the provider
// (of course, in this example we already have it, but
// we want to illustrate how to get layout instances from
// the provider)
IlvGrapherLayout subLayout = provider.getGraphLayout(adapterB);

// Specify node1 (contained in grapherB) as fixed
subLayout.setFixed(node1, true);

// Ask the sublayout to not move the fixed nodes
// This settings holds only to this layout. The
// global settings of the topmost layout are not
// copied to the sublayouts once the sublayouts
// are instanciated.
subLayout.setPreserveFixedNodes(true);
```

```
// Now perform the recursive layout using the Layout Provider.
// The node node1 will be considered as fixed
// by the layout applied to grapherB
...
```

Nesting Several Layouts for Each Graph in the Nested Graph

A more demanding task is to combine several layouts for each graph inside a nested graph. For instance, a node layout followed by a link layout and lastly by a label layout may be needed for each graph.

This case is covered by the section *Recursive Layout* on page 265.

Saving Layout Parameters and Preferred Layouts

> ***Note:*** *To understand this section better, read Using the Graph Model on page 297 first.*

Layout parameters are stored in the layout classes directly, not in the `IlvGrapher`. The advantage of this is that the layout parameters are independent of the graph model. However, the disadvantage is that a layout parameter setting is lost whenever a graph is saved to an `.ivl` file and reloaded later. To overcome this disadvantage, the `IlvGrapherAdapter` class allows you to transfer the graph layout settings to instances of `IlvNamedProperty` that can be stored in `.ivl` files and to recover the settings from these named properties at a later time. (See *Save Parameters to Named Properties* on page 34.)

The following method indicates whether a layout class supports this mechanism:

```
supportsSaveParametersToNamedProperties();
```

It returns `true` if the layout class can transfer the parameter settings to named properties.

Saving Layout Parameters to .ivl Files

The following example shows how to save the `IlvGrapher` including all layout parameter settings into an `.ivl` file. The example assumes that an `IlvGrapher` is attached to three instances `layout1`, `layout2` and `layout3`.

```
IlvGrapherAdapter adapter1 = (IlvGrapherAdapter)layout1.getGraphModel();
IlvGrapherAdapter adapter2 = (IlvGrapherAdapter)layout2.getGraphModel();
IlvGrapherAdapter adapter3 = (IlvGrapherAdapter)layout3.getGraphModel();

// transfer the layout parameters to named properties
if (layout1.supportsSaveParametersToNamedProperties())
  adapter1.saveParametersToNamedProperties(layout1, false);
if (layout2.supportsSaveParametersToNamedProperties())
  adapter2.saveParametersToNamedProperties(layout2, false);
if (layout3.supportsSaveParametersToNamedProperties())
```

```
  adapter3.saveParametersToNamedProperties(layout3, false);

// assume that adapter1, adapter2 and adapter3 work on the same grapher
// save the grapher with all 3 sets of named layout properties to file
grapher.write("abcd.ivl");

// remove the named layout properties because they are no longer needed
adapter1.removeParametersFromNamedProperties();
adapter2.removeParametersFromNamedProperties();
adapter3.removeParametersFromNamedProperties();
```

In this example, *different* grapher adapters of the same `IlvGrapher` are supposed to be attached to `layout1`, `layout2` and `layout3`. If, in fact, the *same* grapher adapter (more precisely, grapher adapters that all work on the same set of nodes and links) is attached to all three layout instances, one call of the `removeParametersFromNamedProperties()` method at the end is sufficient.

Take a look at the mechanism in detail:

```
String saveParametersToNamedProperties(IlvGraphLayout layout,
                                       boolean withDefaults);
```

This method creates named properties for the input layout and transfers the complete layout parameter settings to these properties. The property name is an automatically generated, unique string that is returned. If the flag `withDefaults` is `false`, the created layout properties are persistent only if they contain a setting that is not the default setting. This means that the default values are not stored in the `.ivl` file and the file has a smaller size. If the flag `withDefaults` is `true`, the created properties are always persistent, that is, the default values are stored in the `.ivl` file as well.

The named properties that are created require additional memory. Therefore, it is recommended to remove them as soon as they are no longer needed. To remove the named layout properties, you can use one of the following methods:

- `void removeParametersFromNamedProperties();`

  This method removes all named layout properties from the grapher.

- `void removeParametersFromNamedProperties(String propertyName);`

  This method removes only the named layout properties that match the input property name.

- `void removeParametersFromNamedProperties(Class layoutClass);`

  This method removes all named layout properties that fit the input layout class.

The named layout properties are subclasses of `IlvGraphLayoutGrapherProperty`, `IlvGraphLayoutNodeProperty`, and `IlvGraphLayoutLinkProperty`. See the Reference Manual for details of these classes.

Saving Preferred Layouts to .ivl Files

The class `IlvDefaultLayoutProvider` allows you to specify the layout instances to be used for each graph. The layout provider can then be used for the recursive layout of a nested graph. (For details, see *Laying Out Nested Graphs* on page 306.)

The settings of the preferred layouts, done using the method `setPreferredLayout` of the class `IlvDefaultLayoutProvider`, can be saved to `.ivl` files using the method:

```
boolean
savePreferredLayoutsToNamedProperties(IlvDefaultLayoutProvider
provider, boolean withParameters, boolean withDefaults, boolean
traverse);
```

The method takes the Layout Provider as an argument and several flags:

- `withParameters`: if the flag is `true`, the parameters of the preferred layout instances are saved. In this case it is not necessary to use the method `saveParametersToNamedProperties`. Otherwise, only the name of the class of the preferred layout is saved, without its parameters. Therefore, after loading the `.ivl` file, the preferred layout will have, in this case, the default values for all its parameters.
- `withDefaults`: the flag has the same meaning as for the method `saveParametersToNamedProperties`.
- `traverse`: if the flag is `true`, the method applies recursively to the subgraphs. Otherwise, the method saves only the preferred layout of the grapher adapter on which the method is called.

The following code example shows how to save the preferred layouts to `.ivl` files.

```
IlvGrapher grapherA = new IlvGrapher();
IlvGrapher grapherB = new IlvGrapher();

// fill the graphers with nodes and links;
// grapherB is added as a subgraph of grapherA
grapherA.addNode(grapherB, false);

// Create the grapher adapter for the topmost graph
IlvGrapherAdapter adapterA = new IlvGrapherAdapter(grapherA);

// Get a grapher adapter for the subgraph
IlvGraphModel adapterB = adapterA.getGraphModel(grapherB);

// create the layout provider
IlvDefaultLayoutProvider provider = new IlvDefaultLayoutProvider();

// specify the preferred layouts for each grapher
// (this automatically attaches the layouts)
provider.setPreferredLayout(adapterA, new IlvTreeLayout());
```

```
provider.setPreferredLayout(adapterB, new IlvGridLayout());
...

// Save the settings to named properties
adapterA.savePreferredLayouts(provider, true, false, true);

// Save the nested grapher to an .ivl file
grapherA.write("abcd.ivl")
```

Loading Layout Parameters from .ivl Files

The following example shows how to load and recover the parameters of the three layout instances when the layout settings are stored in an `.ivl` file:

```
// Read the IVL file. This reads all named properties as well.
grapher.read("abcd.ivl");

IlvGrapherAdapter adapter1 = (IlvGrapherAdapter)layout1.getGraphModel();
IlvGrapherAdapter adapter2 = (IlvGrapherAdapter)layout2.getGraphModel();
IlvGrapherAdapter adapter3 = (IlvGrapherAdapter)layout3.getGraphModel();

// Transfer the parameter settings from the named properties to the layouts.
adapter3.loadParametersFromNamedProperties(layout3);
adapter2.loadParametersFromNamedProperties(layout2);
adapter1.loadParametersFromNamedProperties(layout1);

// just to be sure that no named layout properties remain in the memory
adapter1.removeParametersFromNamedProperties();
adapter2.removeParametersFromNamedProperties();
adapter3.removeParametersFromNamedProperties();
```

When reading an `.ivl` file, you usually do not know how many named layout properties are stored in the file. In the previous example, if there are less than three sets of named layout properties stored in the `.ivl` file, any unsuccessful call of `loadParametersFromNamedProperties()` has simply no effect, that is, it does not change the parameters of the corresponding layout. If there are more than three sets, the final calls of the `removeParametersFromNamedProperties()` method guarantee that no memory is wasted by the remaining unused layout properties. As mentioned previously, only one call of `removeParametersFromNamedProperties()` is necessary if only one grapher adapter is attached to all three instances of layout.

Note that the parameters are loaded in the reverse order of that which they are stored. This is not important if all three layout instances are different classes because a layout automatically loads only parameters that fit the layout class. However, if, for example, all three layouts are instances of `IlvTreeLayout`, the last saved set of named properties for any Tree Layout is the first set of named properties that is loaded for a Tree Layout.

To load layout parameters, use one of the following methods:

- `boolean loadParametersFromNamedProperties(IlvGraphLayout layout);`

This method transfers a set of layout parameters from the named layout properties to the input layout. It automatically determines which layout properties fit the input layout. If several sets fit, it transfers the set that was stored last, removes this set of named layout properties from the grapher, and returns `true`. If no set fits and loading cannot be done, it returns `false`.

- `boolean loadParametersFromNamedProperties(IlvGraphLayout layout, String propertyName);`

  This method transfers a set of layout parameters from the layout properties having the input property name to the input layout. It returns `true` if successful, and `false` otherwise.

- `IlvGraphLayout loadParametersFromNamedProperties(IlvGraphLayout layout, String propertyName);`

  This method transfers a set of layout parameters from the layout properties having the input property name to a newly created instance of `IlvGraphLayout`. It returns the new instance. It returns `null` if no set of layout properties with the input name is found.

Loading Preferred Layouts from .ivl Files

The settings of the preferred layouts can be loaded from `.ivl` files using the method:

```
boolean
loadPreferredLayoutsFromNamedProperties(IlvDefaultLayoutProvider
provider, boolean withParameters, boolean traverse);
```

The method takes the Layout Provider as an argument and several flags:

- `withParameters`: if the flag is `true`, the parameters of the preferred layout instances are loaded. In this case it is not necessary to use the method `loadParametersFromNamedProperties`. Otherwise, after loading the `.ivl` file, the preferred layout will have, in this case, the default values for all its parameters.
- `traverse`: if the flag is `true`, the method applies recursively to the subgraphs. Otherwise, the method loads only the preferred layout of the grapher adapter on which the method is called.

This method reads the named properties stored in the file and sets the preferred layout (if any has been stored) on the Layout Provider, using its method `setPreferredLayout`.

The following example shows how to load and recover the preferred layouts (with their parameters) when the preferred layout settings are stored in an `.ivl` file:

```
IlvGrapher grapher = new IlvGrapher();

// Create the grapher adapter for the topmost graph
IlvGrapherAdapter adapter = new IlvGrapherAdapter(grapher);
```

```
// Load the graphers from the .ivl file
grapher.read("abcd.ivl");

// Create the layout provider
IlvDefaultLayoutProvider provider = new IlvDefaultLayoutProvider();

// Load the preferred layouts into the provider
// (layout parameters are also read)
adapter.loadPreferredLayoutsFromNamedProperties(provider, true, true);

// Now the provider can be used to perform the recursive layout

adapter.performLayout(provider, true, true, true);

// Detach the layouts when the provider is no longer needed
provider.detachLayouts(adapter, true);
// Dispose the topmost adapter when no longer needed
adapter.dispose();
```

Additional Information for Expert Users

This section contains useful information for expert users.

Interface Parameters

Some layout classes allow an interface as input parameter. For instance, the method `setNodeBoxInterface` sets an `IlvNodeBoxInterface` (see `IlvLinkLayout`, `IlvShortLinkLayout`, and so on). If an application uses a node box class that implements the node box interface, this can only be stored to an `.ivl` file if the node box class also implements the `IlvPersistentObject` interface. Otherwise, the node box class is not saved to the `.ivl` file.

Compatibility Issues

An `.ivl` file that contains layout properties can be loaded only when the package `ilog.views.graphlayout` and its subpackages are available.

Defining a New Type of Layout

If you develop your own layout algorithms by subclassing `IlvGraphLayout` and want to save the layout parameters of your own layout algorithm in `.ivl` files, you should subclass `IlvGraphLayoutGrapherProperty`, `IlvGraphLayoutNodeProperty`, and `IlvGraphLayoutLinkProperty`. (See *Defining a New Type of Layout* on page 329.) You can find example code by referring to the classes in the *Reference Manual*. You should also override the following methods of `IlvGraphLayout` to return your subclasses:

```
protected IlvGraphLayoutGrapherProperty createLayoutGrapherProperty(
    String name, boolean withDefaults)
{
    return new MyOwnLayoutGrapherProperty(name, this, withDefaults);
}
```

```
protected IlvGraphLayoutNodeProperty createLayoutNodeProperty(
    String name, IlvGraphic node, boolean withDefaults)
{
    return new MyOwnLayoutNodeProperty(name, this, node, withDefaults);
}

protected IlvGraphLayoutLinkProperty createLayoutLinkProperty(
    String name, IlvGraphic link, boolean withDefaults)
{
    return new MyOwnLayoutLinkProperty(name, this, link, withDefaults);
}
```

Further Applications of Layout Properties

The layout properties can be serialized. If you prefer to use the standard serialization mechanism of Java instead of `.ivl` files, it is recommended to use the layout properties as well because it guarantees that only the parameters are serialized and not any temporary data that may exist in the layout instance.

The following code shows an easy way to copy the parameter setting from `layout1` to `layout2`:

```
IlvGrapherAdapter adapter1 = (IlvGrapherAdapter)layout1.getGraphModel();
IlvGrapherAdapter adapter2 = (IlvGrapherAdapter)layout2.getGraphModel();

adapter1.saveParametersToNamedProperties(layout1, true);
adapter2.loadParametersFromNamedProperties(layout2);
```

The following code shows an easy way to undo temporary changes of layout parameters:

```
IlvGrapherAdapter adapter = (IlvGrapherAdapter)layout.getGraphModel();
adapter.saveParametersToNamedProperties(layout, true);

... change layout parameters
... work with changed parameters

// restore the layout parameters as they were before
adapter.loadParametersFromNamedProperties(layout);
```

Using the Filtering Features to Lay Out a Part of an IlvGrapher

> ***Note:*** *To understand this section better, read Using the Graph Model on page 297 first.*

Applications sometimes need to perform the layout algorithm on a subset of the nodes and links of a graph. If the graph is not an `IlvGrapher`, the custom adapter should support the filtering of a graph. (See *Laying Out a Non-JViews Grapher* on page 303.) The methods that are related to the structure of the graph (`getNodes`, `getLinks`, `getNeighbors`, and so on as shown in *Information on the Structure of the Graph* on page 299) must behave just as if the graph has changed in some way. They must take into account only the nodes and links that belong to the part of the graph that must be laid out.

For applications that use `IlvGrapher`, the filtering feature is built into the `IlvGrapherAdapter`. To do this, the `IlvGrapherAdapter` needs a way to know, for each node or link, whether it must be taken into account during the layout. This is the role of the "filter" class, `ilog.views.graphlayout.IlvLayoutGraphicFilter`.

The `IlvLayoutGraphicFilter` class implements the interface `ilog.views.IlvGraphicFilter`, that is, its method:

```
boolean accept(IlvGraphic nodeOrLink)
```

If a filter is specified, the `IlvGrapherAdapter` calls the `accept` method for each node or link whenever necessary. If the method returns `true`, the `IlvGrapherAdapter` considers the node or the link as part of the graph that needs to be laid out. Otherwise, it ignores the node or the link.

To specify a filter on an `IlvGrapherAdapter`, use the following method of the `IlvGrapherAdapter` class:

```
void setFilter(IlvLayoutGraphicFilter filter)
```

To remove the filter, call the `setFilter` method with a `null` argument.

To obtain the filter that has been specified, use the method:

```
IlvLayoutGraphicFilter getFilter()
```

> ***Note:*** *All overridden implementations of the* `accept` *method must respect the following rule: a link cannot be accepted by the filter if any of its end nodes (origin or destination nodes) are not accepted.*

There are two ways to filter an `IlvGrapher`: by layers or by graphic objects. The two methods can be combined.

Filtering by Layers

Inside an `IlvGrapher`, nodes and links can be managed by layers. (See the `ilog.views.IlvManagerLayer` class). ILOG JViews allows you specify that only nodes and links belonging to certain layers have to be taken into account when performing the layout. Use the following methods of the `IlvGrapherAdapter` class:

```
void addLayer(IlvManagerLayer layer)

boolean removeLayer(IlvManagerLayer layer)

boolean removeAllLayers()

boolean isLayerAdded()
```

To get an enumeration of the manager layers to be taken into account during the layout, use the method:

```
Enumeration getLayers()
```

If no layers have been specified or all the specified layers have been removed, all layers in the `IlvGrapher` are used. In this case, the `getLayers` method returns `null`.

When at least one layer is specified, an `IlvLayoutGraphicFilter` is created internally if it has not already been specified using the `setFilter` method. The default implementation of its `accept` method will automatically check whether a node or a link received as an argument belongs to one of the specified layers.

Filtering by Graphic Objects

The nodes and links to be taken into account during the layout can also be filtered individually. To do this, a custom subclass of `IlvLayoutGraphicFilter` must be written. The filtering rules have to be embedded in the overridden implementation of the `accept` method. For example, user properties could be used by an application to "mark" nodes and links that have to be accepted by the filter. The filter class could then be written as follows:

```
public class LayoutFilter
  extends IlvLayoutGraphicFilter
{
  public LayoutFilter()
  {
  }
  public boolean accept(IlvGraphic obj)
  {
    Object prop = obj.getProperty("markedObj");
    if (prop == null)
      return false;
    // accept a link only if its two end-nodes are accepted
    if (obj instanceof IlvLinkImage) {
      IlvLinkImage link = (IlvLinkImage)obj;
      return (link.getFrom().getProperty("markedObj") != null &&
              link.getTo().getProperty("markedObj") != null);
    }
    return true;
  }
}
```

Choosing the Layout Coordinate Space

> ***Note:*** *To understand this section better, read Using the Graph Model on page 297 first.*

Graph layout algorithms have to deal with the geometry of the graph, that is, the position and shape of the nodes and links. In addition, they must deal with the layout of an `IlvGrapher`. The nodes of an ILOG JViews `IlvGrapher` can be any graphic object, that is, any subclass of `IlvGraphic`. The position and size of the nodes are given by their

`boundingBox(IlvTransformer t)` method and usually depend on the transformer used for their display. Therefore, when an `IlvGrapher` has to be laid out, the geometry of the grapher must be considered for a given value of the transformer.

The most natural transformer value that could be chosen is the "identity" transformer. An identity transformer has no translation, zoom, or rotation factors. In terms of ILOG JViews, this would mean that the geometry of the `IlvGrapher` would be considered in the manager coordinates, not in the manager view coordinates (transformed coordinates). However, the special case of *nonzoomable* graphic objects must be taken into account. For this case, the idea of simply using the geometry of the grapher in manager coordinates is not pertinent.

A Special Case: Nonzoomable Graphic Objects

A graphic object is said to be *zoomable* if its bounding box follows the zoom level. Otherwise, the object is *nonzoomable*. (To know whether a graphic object is zoomable, use its `boolean zoomable()` method, or check its documentation.)

If all the nodes and links of an `IlvGrapher` are zoomable graphic objects, a layout obtained on the basis of the graph geometry in manager coordinates will "look" the same for any value of the transformer used for the display. Simply speaking, the drawing of the graph will just be zoomed, or translated.

When at least one nonzoomable graphic object is used as a node in an `IlvGrapher`, the geometry of the grapher in manager coordinates can no longer be used. When drawn with different transformer values (for instance, at different zoom levels), the same `IlvGrapher` can look very different.

Reference Transformers

When a grapher contains nonzoomable graphic objects, it may not be appropriate to deal with the geometry of the `IlvGrapher` based on the graph objects bounding boxes systematically computed for an identity transformer (manager coordinates). To ensure that the drawing of the laid-out graph is always correct, including the case where nonzoomable graphic objects are present, the transformer used for the display must be considered. Generally speaking, the layout of an `IlvGrapher` depends on the transformer.

Instead of dealing with the issue of zoomable/nonzoomable objects and transformers at the level of the layout algorithms, the ILOG JViews Graph Layout module delegates this task to the `IlvGrapherAdapter`.

Layout algorithms interact with the geometry of the graph using generic methods of the graph model (`IlvGraphModel`), such as `boundingBox(Object nodeOrLink)`. The distinction between zoomable and nonzoomable objects, and the notion of transformer (`IlvTransformer`), are outside this level of the layout framework. The layout algorithms consider the geometry of the graph exactly as it is provided by the graph model. From the point of view of the layout algorithms, the problem of zoomable and nonzoomable objects is

completely transparent. Therefore, when writing a layout algorithm, you do not need to be concerned with that.

In the case of an `IlvGrapher`, the `IlvGrapherAdapter` may need to compute the geometry of the graph for a given transformer. This is what we call the *reference transformer*. Usually, the reference transformer is the transformer that is currently being used for the display of the `IlvGrapher`.

How a Reference Transformer is Used

For a simple example of how a reference transformer is used, consider the `boundingBox(Object nodeOrLink)` method. This abstract method of the `IlvGraphModel` class is implemented in the `IlvGrapherAdapter`. To compute the bounding box, it calls the `boundingBox(IlvTransformer t)` method of the graphic object that it receives as an argument. However, it does not handle zoomable objects and nonzoomable objects in the same way.

If the graphic object is zoomable, the `boundingBox(Object nodeOrLink)` method of the `IlvGrapherAdapter` returns the bounding box in manager coordinates by calling `IlvGraphic.boundingBox(null)`.

If the graphic object is nonzoomable, the `boundingBox(Object nodeOrLink)` method computes the bounding box according to the reference transformer and returns a rectangle obtained by applying the inverse transformation to this rectangle. (See the `IlvTransformer.inverse(IlvRect rect)` method.)

The geometry of the `IlvGrapher` is computed in such a manner that the resulting drawing inside an `IlvManagerView` using the reference transformer will look fine.

Reference Views

Optionally, an `IlvManagerView` can be specified as a *reference view* for the `IlvGrapherAdapter`. If a reference view is specified, its current transformer (at the moment when the layout is started) is automatically used as the reference transformer. Usually, applications use the same manager view that is used for the display of the `IlvGrapher` as the reference view (but this is not mandatory).

To specify the reference view, use the following method:

```
void setReferenceView(IlvManagerView view)
```

To get the current reference view, use the method:

```
IlvManagerView getReferenceView()
```

If no view has been specified as the reference view, the method returns `null`.

Specifying a Reference Transformer

A reference transformer can be specified explicitly using the method:

```
void setReferenceTransformer(IlvTransformer transformer)
```

The current reference transformer is returned by the method:

```
IlvTransformer getReferenceTransformer()
```

In most cases, it is not necessary to specify a reference transformer because the last method automatically chooses it according the following rules:

- If a reference transformer is specified, the specified transformer is returned.
- If a reference view has been specified, the transformer of the reference view is returned.
- If the `IlvGrapher` attached to the `IlvGrapherAdapter` has at least one manager view, the transformer of the first manager (as returned by the method `IlvManager.getViews()`) is returned.

The only cases where you may need to specify a reference transformer or a reference view are the following:

- The `IlvGrapher` contains nonzoomable objects (that is, the layout cannot be correctly computed independently of the transformer used for drawing the graph) and more than one manager view is attached to the grapher.
- The `IlvGrapher` contains nonzoomable objects and you want to perform the layout without attaching a manager view to the grapher. (Therefore, the default rule for choosing the current transformer of the first manager view as the reference transformer cannot be applied.)

If a grapher containing nonzoomable objects is displayed simultaneously in several views, you can use the `setReferenceView` method to indicate the view for which you want the drawing of the graph to be optimal.

If you specified a reference transformer but want to reset this setting and go back to the default behavior, call the method `setReferenceTransformer` with a `null` argument.

Note that if you override the `setReferenceTransformer` method, you must call `super.setReferenceTransformer` to notify the `IlvGrapherAdapter` that the reference transformer has changed.

Note also that a call to the `setReferenceView` method overrides the effect of a call to the `setReferenceTransformer` method. In the same way, a call to the `setReferenceTransformer` method overrides the effect of a call to the `setReferenceView` method.

Specifying the Coordinates Mode

By default, the `IlvGrapherAdapter` considers the geometry of the nodes and links of an `IlvGrapher` in a special coordinate space which is appropriate for most of the cases. In some situations, it can be useful to specify a different coordinate space.

To specify the coordinate space, the class `IlvGrapherAdapter` provides the following method:

```
void setCoordinatesMode(int mode)
```

The valid values for `mode` are:

- `IlvGraphLayout.MANAGER_COORDINATES`

  The geometry of the graph is computed using the coordinate space of the manager (that is, the `IlvGrapher` encapsulated by the adapter) without applying any transformation.

  This mode should be used if you visualize the graph at zoom level 1, or you do not visualize it at all, or the grapher contains only fully zoomable objects. In all these cases there is no need to take the transformer into account during the layout.

  Note that in this mode the dimensional parameters of the layout algorithms are considered as being specified in manager coordinates. The reference transformer and the reference view are not used.

- `IlvGraphLayout.VIEW_COORDINATES`

  The geometry of the graph is computed in the coordinate space of the manager view. More exactly, all the coordinates are transformed using the current reference transformer.

  This mode should be used if you want the dimensional parameters of the layout algorithms to be considered as being specified in manager view coordinates.

- `IlvGraphLayout.INVERSE_VIEW_COORDINATES`

  The geometry of the graph is computed using the coordinate space of the manager view and then applying the inverse transformation using the reference transformer. This mode is equivalent to the "manager coordinates" mode if the geometry of the graphic objects strictly obeys the transformer. (A small difference may exist because of the limited precision of the computations.)

  On the contrary, if some graphic objects are either nonzoomable or semizoomable (for example, links with a maximum line width), this mode gives different results than the manager coordinates mode. These results are optimal if the grapher is visualized using the same transformer as the one taken into account during the layout.

  Note that in this mode the dimensional parameters of the layout algorithms are considered as being specified in manager coordinates.

The default mode is `IlvGraphLayout.INVERSE_VIEW_COORDINATES`.

To obtain the current choice, use the following method:

```
long getCoordinatesMode()
```

The coordinates mode can also be specified directly on the layout instances. For details, see *Coordinates Mode* on page 24.

Releasing Resources Used During the Layout

Various objects need to be created during the layout process. Most commonly, these are layout instances (subclasses of `IlvGraphLayout`) and grapher adapters (subclasses of `IlvGrapherAdapter`) or other adapters (subclasses of `IlvGraphModel`). For recursive layout (see *Laying Out Nested Graphs* on page 306), layout providers (subclasses of `IlvDefaultLayoutProvider`) may also be instantiated. Finally, some of the layout parameters are internally stored as property objects attached to the grapher object or to its nodes and links.

To ensure that all these allocated objects are correctly released once they are no longer needed (such obsolete objects can be garbage collected and memory leaks (in the Java sense) avoided), the following rules must be respected:

1. When a layout instance instantiated by your code is no longer useful, call the method `detach()` on it to ensure that no grapher or graph model is still attached to it. Note that you can freely reuse a layout instance once the previously attached model has been detached.
2. Layout parameters that are specific to a node or a link are cleaned when calling `IlvGraphLayout.detach()`. This cleaning is done only for nodes and links that are still in the grapher when the `detach()` method is called. If per-node or per-link parameters have been specified and the node or the link needs to be removed before the `detach()` method can be called, you can call the methods `cleanNode` or `cleanLink` of the class `IlvGraphLayout` to perform the cleaning for the node or the link. However, you only need to do so if the removed node or link is reused by your code after removal. Otherwise, if your code does not keep any reference to it, the node or link will be garbage collected anyway, together with the property objects eventually stored by the layout.
3. When a grapher adapter (or other graph models) instantiated by your code is no longer useful, call the method `dispose()` on it to ensure that the resources it has used are released. Note that an adapter (or graph model) must not be used once it has been disposed.

When a layout provider (an instance of `IlvDefaultLayoutProvider`) instantiated by your code is no longer useful, call the method `detachLayouts(model, true)` on it, passing as arguments the graph models that have been used for performing a recursive layout with this provider.

Defining a New Type of Layout

If the layout algorithms provided in ILOG JViews do not meet your needs, you can develop your own layout algorithms by subclassing `IlvGraphLayout`.

When a subclass of `IlvGraphLayout` is created, it automatically fits into the generic layout framework of ILOG JViews and benefits from its infrastructure: generic parameters (see *Layout Parameters and Features in IlvGraphLayout* on page 20, notification of progress (see *Using Event Listeners* on page 294), capability to lay out any graph object using the generic graph model (see *Using the Graph Model* on page 297), capability to apply the layout separately for the connected components of a disconnected graph (see *Laying Out Connected Components of a Disconnected Graph* on page 304), capability to lay out nested graphs (see *Laying Out Nested Graphs* on page 306), and so on.

Sample Code

To illustrate the basic ideas for defining a new layout, the following simple example shows a possible implementation of the simplest layout algorithm, the Random Layout. The new layout class is called `MyRandomLayout`.

The following shows the skeleton of the class:

```
public class MyRandomLayout
  extends IlvGraphLayout
{
  public MyRandomLayout()
  {
  }

  public MyRandomLayout(MyRandomLayout source)
  {
    super.source(source);
  }

  public IlvGraphLayout copy()
  {
    return new MyRandomLayout(this);
  }

  protected void layout(boolean redraw)
  {
    ...
  }
}
```

The constructor with no arguments is empty. The `copy` constructor and the `copy` method are implemented; they are used when laying out a nested graph (see *Laying Out Nested Graphs* on page 306). Then, the abstract method `layout(boolean)` of the base class is implemented as follows:

```
  protected void layout(boolean redraw)
  {
    // obtain the graph model
    IlvGraphModel graphModel = getGraphModel();

    // obtain the layout report
    IlvGraphLayoutReport layoutReport = getLayoutReport();

    // obtain the layout region
    IlvRect rect = getCalcLayoutRegion();
    float xMin = rect.x;
    float yMin = rect.y;
    float xMax = rect.x + rect.width;
    float yMax = rect.y + rect.height;

    // initialize the random generator
    Random random = (isUseSeedValueForRandomGenerator()) ?
        new Random(getSeedValueForRandomGenerator()) :
        new Random();

    // browse the objects in the grapher
    Enumeration nodes = graphModel.getNodes();
    while (nodes.hasMoreElements()) {
        Object node = nodes.nextElement();

        // skip fixed nodes
        if (isPreserveFixedNodes() && isFixed(node)))
            continue;

        // compute coordinates
        float x = xMin + (xMax - xMin) * random.nextFloat();
        float y = yMin + (yMax - yMin) * random.nextFloat();

        // move the node to the computed position
        graphModel.moveNode(node, x, y, redraw);

        // notify listeners on layout events
        callLayoutStepPerformedIfNeeded();
    }

    // set the layout report code
    layoutReport.setCode(IlvGraphLayoutReport.LAYOUT_DONE);
}
...
```

Note that the `layout` method is `protected`, which is the access type of the method in the base class. This will not prevent a user outside the package containing the class from performing the layout because it is started using the public method `performLayout`.

Steps for Implementing the Layout Method

In our example, the `layout` method is implemented using the following main steps:

1. Obtain the graph model (`getGraphModel()` on the layout instance).

   ```
   IlvGraphModel graphModel = getGraphModel();
   ```

2. Obtain the instance of the layout report that is automatically created when the `performLayout` method from the superclass is called (`getLayoutReport()` on the layout instance).

   ```
   IlvGraphLayoutReport layoutReport = getLayoutReport();
   ```

3. Obtain the layout region parameter to compute the area where the nodes will be placed.

   ```
   IlvRect rect = getCalcLayoutRegion();
   ```

4. Initialize the random generator.

   ```
   Random random = (isUseSeedValueForRandomGenerator()) ?
           new Random(getSeedValueForRandomGenerator()) :
           new Random();
   ```

   (For information on the seed value parameter, see *Random Generator Seed Value* on page 33.)

5. Get an enumeration of the nodes (`getNodes()` on the graph model instance).

   ```
   Enumeration nodes = graphModel.getNodes();
   ```

6. Browse the nodes, skipping fixed nodes (`isFixed(node)` on the layout instance) if asked by the user (`isPreserveFixedNodes()` on the layout instance).

   ```
   while (nodes.hasMoreElements()) {
      Object node = nodes.nextElement();
   ...
   ```

   (For details on fixed nodes, see *Preserve Fixed Nodes* on page 32).

7. Move each node to the newly computed coordinates inside the layout region (`graphModel.moveNode`).

   ```
   graphModel.moveNode(node, x, y, redraw);
   ```

8. Notify the listeners on layout events that a new node was positioned (`callLayoutStepPerformedIfNeeded()` on the layout instance). This allows the user to implement, for example, a progress bar if a layout event listener was registered on the layout instance.

   ```
   callLayoutStepPerformedIfNeeded();
   ```

   (For details on event listeners, see *Using Event Listeners* on page 294.)

9. Finally, set the code in the layout report.

   ```
   layoutReport.setCode(IlvGraphLayoutReport.LAYOUT_DONE);
   ```

Of course, depending on the characteristics of the layout algorithm, some of these steps may be different or unnecessary, or other steps may be needed.

Depending on the particular implementation of your layout algorithm, other methods of the `IlvGraphLayout` class may need to be overridden. For instance, if your subclass supports some of the generic parameters of the base class, you must override the `supports[ParameterName]` method (see *Layout Parameters and Features in IlvGraphLayout* on page 20). For further information about the API of the class `IlvGraphLayout`, please refer to the Reference Manual of this class..

> ***Note:*** *If you want to save the layout parameters of your new layout algorithm in* `.ivl` *files, you should override the methods* `createLayoutGrapherProperty`, `createLayoutNodeProperty`, `createLayoutLinkProperty`, *and subclass* `IlvGraphLayoutGrapherProperty`, `IlvGraphLayoutNodeProperty`, *and* `IlvGraphLayoutLinkProperty`. *See Saving Layout Parameters and Preferred Layouts on page 315 for more explanation.*

Questions and Answers about Using the Layout Algorithms

Table 5.2 provides some helpful suggestions for using the layout algorithms. You may find some answers to questions that come up when using the Graph Layout module.

Table 5.2 *Questions and Answers about the Layout Algorithms*

Question	Answer
I perform the layout and nothing happens (no node is moved). Why?	One possible reason may be: the layout algorithms provided in ILOG JViews are all designed to do nothing, by default, if no change occurred in the graph since the last time the layout was performed successfully on the same graph. A change means that a node was moved, or a node or link was added, removed, or reshaped. Note that you can force the layout to be performed again, even if no change occurred, by calling the `IlvGraphLayout.performLayout(boolean force, boolean redraw)` method with a `true` value for the force argument. (In the Composer demonstration, you can choose this option in the Options menu.) Another possible reason may be: an error or a special case occurred during the layout. First, you should check whether the `performLayout` method has thrown an exception. If no exception was thrown, call the `getCode()` method on the instance of the layout report returned by the `performLayout` method. Check this value with respect to the documentation of the appropriate layout report class. (For details, see *Using a Layout Report* on page 292.)
With the Uniform Length Edges algorithm, after having performed the layout once, I don't see any movement even if I use the force layout option. Why?	The reason is probably that the first time you performed the layout, the algorithm reached the convergence. When the layout is performed again, it detects that the convergence has been already reached and stops. If you really want to continue working, for instance in order to "declutter" a particular part of the graph, you may need to move one or several nodes in order to change the initial configuration. (The algorithm is dependent on the initial configuration.)

Table 5.2 *Questions and Answers about the Layout Algorithms (Continued)*

Question	Answer
After performing the layout, the graph is laid out far from its initial position. Why?	Most of the layout algorithms use a layout region parameter to control the size and position of the layout. (For details, see *Layout Region* on page 25.) Depending on the value of this parameter, the nodes may be moved far from their initial positions. To know whether a layout algorithm is designed to use a layout region parameter, check the documentation to see if the layout class overrides the `supportsLayoutRegion()` method of the base class in order to return `true`. Other algorithms have a different mechanism that allows you to specify the desired location of the layout. It may happen that the default value of the location parameter is such that the graph is laid out far from its initial position.
When I use certain layout algorithms on certain graphs, there are overlapping nodes. Why and what can I do?	One possible reason may be related to the different ways layout algorithms deal with the size of the nodes: -The Topological Mesh and Spring Embedder algorithms are not able to explicitly take into account the size of the nodes. - The Tree, Hierarchical, Bus, and Grid algorithms always avoid overlapping nodes. (The Link algorithm does not move the nodes. It only reshapes the links such that the crossings and overlaps are reduced. The size of the nodes is taken into account.) - The Uniform Length Edges algorithm (with the option “Respect Node Sizes” enabled) and the Circular algorithm, in many cases, succeed in avoiding overlapping nodes. In any case, if the layout algorithm supports the layout region mechanism (see *Layout Region* on page 25), you should try to increase the size of the layout region. For example, if your graph contains hundreds of nodes, it is not reasonable to use a small layout region, such as 600x600. There will be not enough space for all the nodes. You should try a larger layout region, for example 5000x5000. The optimal size of the layout region depends, of course, not only on the number of nodes, but also on their size. If the nodes are relatively large with respect to the size of the layout region, it may be necessary to adjust some of the parameters (for instance, the preferred link length for the Uniform Length Edges Layout).

Table 5.2 *Questions and Answers about the Layout Algorithms (Continued)*

Question	Answer
In some networks, there are two (or more) subnetworks that are not connected. How will this affect the layout algorithms provided in ILOG JViews?	This depends on the layout class you use: - `IlvTopologicalMeshLayout`: It will work on the connected component of the graph that contains the starting node. (You can specify this node as a parameter.) If the "starting node" is not specified, it is automatically chosen in an arbitrary way. The nodes of the other "connected components" will not be moved. You may want to perform the layout separately on each connected component using different layout regions and starting node settings. This is what you get automatically when you enable the "layout of connected components" parameter. (See *Layout of Connected Components* on page 25.) - `IlvSpringEmbedder`, `IlvUniformLengthEdges`: These algorithms support disconnected graphs, but usually it is better to rely on the automatic "layout of connected components" parameter. (See *Layout of Connected Components* on page 25.) - `IlvBusLayout`: It will work on the "connected component" of the graph that contains the "bus object." (You must specify the bus object as a parameter.) The other nodes that are not connected to the bus will not be moved. You may need to perform the layout separately on each connected component. This is what you get automatically when you enable the "layout of connected components" parameter. (See *Layout of Connected Components* on page 25.) - `IlvCircularLayout`, `IlvHierarchicalLayout`, `IlvTreeLayout`: They have built-in support for disconnected graphs. Alternatively, you can use the automatic support from the base class. (See *Layout of Connected Components* on page 25.) - `IlvLinkLayout`, `IlvGridLayout`, `IlvRandomLayout`: These algorithms support both connected and disconnected graphs. Their behavior is the same for both categories of graphs.

Table 5.2 *Questions and Answers about the Layout Algorithms (Continued)*

Question	Answer
There are some attributes of the network that we know about (for instance, we know what the core switch is and what the center should be). Are such attributes taken into account by the layout algorithm?	It depends on the layout algorithm. - The Circular Layout is designed to allow you to specify information about the physical topology of the network. You can specify which nodes belong to the same cluster (ring or star), the order of the nodes on the cluster, and which node is the center of a star cluster. - In the Tree Layout, you can specify the root node. - In the Bus Layout algorithm, you can specify the bus object. - In the Hierarchical Layout algorithm, you can specify node position indices and level indices, as well as relative positioning constraints.

Table 5.2 *Questions and Answers about the Layout Algorithms (Continued)*

Question	Answer
If I use ILOG JViews on different computers and/or with different Java Virtual Machines (JVM), I sometimes get different layouts for the same graph and with the same parameters. Why?	There are two possible reasons: 1. Different computers and JVMs may be slower or faster. If the layout algorithm you use stops the computation when the specified allowed time has elapsed, a slower computer or JVM will cause the computation to stop earlier. That may be the cause of different results. This may happen even with the same computer and JVM if the charge of the computer is increased. You may need to increase the allowed time specification when running on a slower computer or JVM. 2. If you use a layout algorithm that uses the random generator and if you use the default option for the seed value (that is, the system clock is used), you get different results for each successive run of the layout on the same graph. This allows you to obtain alternative results and to chose the one you prefer. If you want to prevent different results for successive runs, you can specify a constant seed value.
I use the Link Layout algorithm to lay out the links (representing routes) of a network of graphical objects (towns) geo-positioned on a cartographical map. When several links connect to the same side of a node, they overlap, while I expect them to respect the “link offset” (or the “grid size”) parameter of the Link Layout. Why?	Some dimensional parameters of the layout algorithms need to be chosen with respect to the size of the nodes. This is the case of the “link offset” and the “bypass distance” parameters for the Short Link Layout and the grid size for the Long Link Layout. Indeed, their default values are not appropriate when the nodes are very large. Often, nodes placed on a map, for instance a world map, have a very large size. Compared to this size, the default values of the parameters are so small that they appear to be zero. The solution is to increase the values of the dimensional parameters, taking into account the size of the nodes. If different nodes have different sizes, either the medium or the largest size of the nodes can be used to compute the parameters as a fraction of this size.

C H A P T E R

6

Automatic Label Placement

This chapter describes the label placement algorithms of the ILOG JViews Graph Layout module. A label placement algorithm is not a graph layout algorithm in the sense that it does not use the `IlvGraphModel` and it is not a subclass of `IlvGraphLayout`. However, the labeling framework has many similarities to the graph layout framework and is included in this *Graph Layout User's Manual*. The following topics are covered:

- *Getting Started with Labeling*
- *The Base Classes: IlvLabelingModel and IlvLabelLayout*
- *The Annealing Label Layout*
- *The Random Label Layout*
- *Using Advanced Features*
- *Defining Your Own Labeling Model*

Getting Started with Labeling

This section provides information to get started using the ILOG JViews Label Layout framework.

- An Introduction explains the goal and the limitations of automatic label placement.
- A set of Basic Steps illustrates the basic steps to perform a label placement with the Annealing Label Layout algorithm.
- The Sample Java Application included in the release is also discussed.

Introduction to Automatic Label Placement

Usually, a *label* is a text or decoration that should be placed close to some graphic object because the label denotes the meaning of the graphic object. The label should not be overlapped by obstacles because this would make it unreadable. If there are many potential positions for one label, the automatic label placement algorithm should find the best position for the label, so that it is close to its graphic object, not too close to unrelated objects, and not overlapped by any obstacle.

Labels occur in many application areas: diagrams, geographic maps, charts, and so on. ILOG JViews labeling can be applied to all these areas, that is, it is not restricted to placing labels in a graph. For instance, if you want to place labels at the cities in a geographic map, you can use the Annealing Label Layout algorithm with appropriate point label descriptors. For simplicity, we focus in this Getting Started section on placing labels at links in a graph.

In ILOG JViews, graphic objects (subclasses of `IlvGraphic`) are contained in a manager (`IlvManager`), and graphs are managed by graphers (`IlvGrapher`, which is a subclass of `IlvManager`). Because the ILOG JViews Label Layout algorithms can be applied not only to graphs but also to any graphic objects, the algorithms work on `IlvManager`. Consequently a label layout algorithm is *no*t a subclass of `IlvGraphLayout` and *does not use* an `IlvGraphModel`. ILOG JViews provides a label layout framework that is (despite many similarities) completely independent from the graph layout framework.

The label layout framework distinguishes simply between:

- graphic objects called *labels* that must be placed, and
- graphic objects called *obstacles* that must not move but occupy space that is not available for the labels.

A label should not overlap any obstacle or other label. In fact, this strict condition is unfeasible if there is not enough free space for the labels; in this situation the Annealing Label Layout algorithm tries to reduce the amount of overlaps.

The ILOG JViews Label Layout framework is built on top of the ILOG JViews core library and can coexist with the ILOG JViews Graph Layout framework. For details on the core library, see the *ILOG JViews Graphics Framework User's Manual*.

> ***Note:*** *The label layout framework allows you to add layout capabilities to applications that do not use the ILOG JViews manager. For details see Creating a New Labeling Model on page 389.*

Basic Steps to Perform a Label Placement

To use the Annealing Label Layout algorithm provided by the Graph Layout module of ILOG JViews, you will usually perform the following steps:

1. Create a manager object (`IlvManager`) and fill it with obstacles and labels. For instance, if you want to place labels at the links of a graph, create a grapher and fill it with nodes and links (the obstacles) and additionally with the labels that should be placed at the links. In this case, the labels should be added using `IlvManager.addObject` instead of `IlvGrapher.addNode` because the labels should not be nodes at the same time.
2. Create an instance of the Annealing Label Layout algorithm.
3. Declare a handle for the corresponding layout report. The layout report is an object in which the layout algorithm stores information about its behavior. For details, see *The Label Layout Report* on page 351.
4. Attach the manager to the layout instance. (We assume that the labels are subclasses of `IlvLabel` or `IlvZoomableLabel`. If they are not, the label model can be extended as illustrated in *The Labeling Model* on page 346.)
5. For each label, set a label descriptor. The label descriptor describes the conditions for the placement of the label. For instance, if a label should be placed close to the source or destination node of a link, use an `IlvAnnealingPolylineLabelDescriptor` with the corresponding options. For details, see *Label Descriptors* on page 359.
6. Modify the default settings for the layout parameters, if needed.
7. Call the performLayout method.
8. Read and display information from the layout report.
9. When the layout instance is no longer needed, detach the manager from the layout instance.

An application that illustrates these steps is provided in the section *Sample Java Application* on page 342.

Sample Java Application

You can use this application as an example to get started with the label layout algorithms of the Graph Layout module. It is included as a sample Java application with this release. This section describes how to compile and run the application and provides the sample code.

The source code of the application is named `LayoutSample2.java` and can be found at the location:

```
<installdir>/doc/usermansrc/graphlayout/LayoutSample2.java
```

To compile and run the example, do the following:

1. Go to the `graphlayout` directory at the above path. (On Windows 98-2000-NT, you must open a DOS Console).
2. Set the `CLASSPATH` variable to the ILOG JViews library and the current directory.

   On Windows 98-2000-NT-XP: `.;<installdir>\classes\jviewsall.jar`

   On Unix: `.:<installdir>/classes/jviewsall.jar`

3. Compile the application:

   ```
   javac LayoutSample2.java
   ```

4. Run the application:

   ```
   java LayoutSample2
   ```

The `LayoutSample2.java` contains the following code:

```
// the ILOG JViews Graphic Framework
import ilog.views.*;
import ilog.views.graphic.*;

// the ILOG JViews Label Layout Framework
import ilog.views.graphlayout.labellayout.*;

// the ILOG JViews Annealing Label Layout
import ilog.views.graphlayout.labellayout.annealing.*;

// the Java AWT package
import java.awt.*;


/**
 * A very simple example for the use of a Layout Algorithm.
 */
public class LayoutSample2
{
  public static final void main(String[] arg)
  {
    // Create the manager instance (subclass of IlvManager). Since we want
```

```
// to place link labels, we use a grapher in this example.
IlvGrapher grapher = new IlvGrapher();

// Create the manager view instance
IlvManagerView view = new IlvManagerView(grapher);
view.setBackground(Color.white);

// An AWT Frame to display
Frame frame = new Frame("Label Layout Sample");

// Put the manager view inside an AWT Frame and show it
frame.add(view);
frame.setSize(400, 400);
frame.setVisible(true);

// Fill the grapher with nodes and links
IlvReliefRectangle node1 = new IlvReliefRectangle(
                                  new IlvRect(0f, 0f, 50f, 50f));
IlvReliefRectangle node2 = new IlvReliefRectangle(
                                  new IlvRect(200f, 0f, 50f, 50f));
IlvReliefRectangle node3 = new IlvReliefRectangle(
                                  new IlvRect(0f, 200f, 50f, 50f));
IlvReliefRectangle node4 = new IlvReliefRectangle(
                                  new IlvRect(200f, 200f, 50f, 50f));
grapher.addNode(node1, false);
grapher.addNode(node2, false);
grapher.addNode(node3, false);
grapher.addNode(node4, false);

// set some nice colors
setNodeColors(node1, node2, node3, node4);

IlvLinkImage link1 = new IlvLinkImage(node1, node2, true);
IlvLinkImage link2 = new IlvLinkImage(node1, node3, true);
IlvLinkImage link3 = new IlvLinkImage(node2, node4, true);
IlvLinkImage link4 = new IlvLinkImage(node4, node3, true);
IlvLinkImage link5 = new IlvLinkImage(node1, node4, true);

grapher.addLink(link1, false);
grapher.addLink(link2, false);
grapher.addLink(link3, false);
grapher.addLink(link4, false);
grapher.addLink(link5, false);

// set some nice colors
setLinkColors(link1, link2, link3, link4, link5);

// Add labels. Labels are neither "nodes" nor "links", hence add them as
// objects. Since we perform layout later, the initial position doesn't
// play a role.

IlvPoint p = new IlvPoint(0f,0f);
IlvZoomableLabel label1 = new IlvZoomableLabel(p, "Label1");
```

```
IlvZoomableLabel label2 = new IlvZoomableLabel(p, "Label2");
IlvZoomableLabel label3 = new IlvZoomableLabel(p, "Label3");
IlvZoomableLabel label4 = new IlvZoomableLabel(p, "Label4");
IlvZoomableLabel label5 = new IlvZoomableLabel(p, "Start Label");
IlvZoomableLabel label6 = new IlvZoomableLabel(p, "End Label");

grapher.addObject(label1, false);
grapher.addObject(label2, false);
grapher.addObject(label3, false);
grapher.addObject(label4, false);
grapher.addObject(label5, false);
grapher.addObject(label6, false);


// Declare a handle for the layout instance
IlvAnnealingLabelLayout layout = new IlvAnnealingLabelLayout();

// Declare a handle for the layout report
IlvLabelLayoutReport layoutReport = null;

// Attach the manager to the layout instance
layout.attach(grapher);

// For each label, set a label descriptor that specifies: label1 belongs
// to link1, label2 belongs to link2, and so on. link5 has 2 labels.

layout.setLabelDescriptor(
      label1,
      new IlvAnnealingPolylineLabelDescriptor(
            label1, link1, IlvAnnealingPolylineLabelDescriptor.CENTER));
layout.setLabelDescriptor(
      label2,
      new IlvAnnealingPolylineLabelDescriptor(
            label2, link2, IlvAnnealingPolylineLabelDescriptor.CENTER));
layout.setLabelDescriptor(
      label3,
      new IlvAnnealingPolylineLabelDescriptor(
            label3, link3, IlvAnnealingPolylineLabelDescriptor.CENTER));
layout.setLabelDescriptor(
      label4,
      new IlvAnnealingPolylineLabelDescriptor(
            label4, link4, IlvAnnealingPolylineLabelDescriptor.CENTER));
layout.setLabelDescriptor(
      label5,
      new IlvAnnealingPolylineLabelDescriptor(
            label5, link5, IlvAnnealingPolylineLabelDescriptor.START));
layout.setLabelDescriptor(
      label6,
      new IlvAnnealingPolylineLabelDescriptor(
            label6, link5, IlvAnnealingPolylineLabelDescriptor.END));

// Modify the layout parameters, if needed
layout.setLabelOffset(10);
```

```
    layout.setObstacleOffset(5);

    // Perform the layout and get the layout report
    layoutReport = layout.performLayout();

    // Print information from the layout report (optional)
    System.out.println("layout done in " +
                       layoutReport.getLayoutTime() +
                       " millisec., code = " +
                       layoutReport.getCode());

    // Fit the graph in the window
    view.fitTransformerToContent();
    // Redraw the grapher
    grapher.reDraw();

    // Detach the grapher from the layout instance
    layout.detach();
  }

  // ... further methods ...
}
```

The sample Java application produces the graph shown in Figure 6.1.

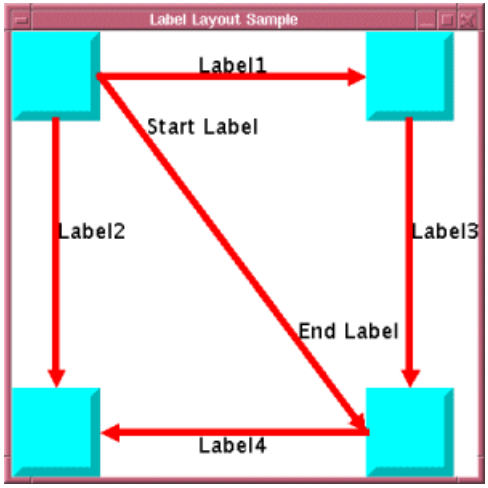


Figure 6.1 *Output from Sample Java Application*

The Base Classes: IlvLabelingModel and IlvLabelLayout

The base classes for automatic label placement are `IlvLabelLayout`, the base class for labeling algorithms, and `IlvLabelingModel`, a base class for performing many of the generic operations common to all labeling algorithms.

> ***Note:*** *Before reading this section, you should be familiar with the IlvGraphLayout and IlvGraphModel classes (see The Base Class: IlvGraphLayout on page 17 and Using the Graph Model on page 297). Many of the concepts for the labeling layout mechanism and labeling model are similar, and we mention the common features here without repeating all identical details.*

- *The Labeling Model* - discusses how to use the `IlvLabelingModel` class.
- *The Label Layout Base Class* - introduces the `IlvLabelLayout` class.

The Labeling Model

The `ilog.views.graphlayout.labellayout.IlvLabelingModel` class defines a suitable, generic API to position labels automatically by an ILOG JViews label layout algorithm. Its purpose is similar to the `IlvGraphModel` for the graph layout framework.

`IlvLabelingModel` is an abstract base class that allows you to implement an adapter to your own data structures for the labeling algorithm (see *Defining Your Own Labeling Model* on page 384). The labeling model of an attached layout can be obtained by:

```
layout.getLabelingModel();
```

If labels and obstacles are contained in an `IlvManager`, you do not need to implement an adapter. You can use the default labeling model provided by ILOG JViews (the class `ilog.views.graphlayout.labellayout.IlvDefaultLabelingModel`).

The default labeling model is suitable if the labels are upright rectangular objects and the overlap calculation considers their bounding boxes. The default labeling model may not be suitable if the labels have a nonrectangular shape that leaves large parts of the label's bounding box empty, because the overlap detection will wrongly assume that the empty space inside the bounding box causes overlaps.

The default labeling model considers subclasses of `IlvLabel` and `IlvZoomableLabel` as labels and all other objects as obstacles. This is the most common case. However, you can redefine this meaning:

- If a graphic should be treated as label even though it is not a subclass of `IlvLabel` and `IlvZoomableLabel`, call:

  ```
  defaultLabelingModel.setLabel(graphic, true);
  ```

- If an instance of `IlvLabel` or `IlvZoomableLabel` should not be treated as a label but as an immovable obstacle, call:

  ```
  defaultLabelingModel.setObstacle(graphic, true);
  ```

- If a graphic should be ignored by the labeling model, call:

  ```
  defaultLabelingModel.setLabel(graphic, false);
  defaultLabelingModel.setObstacle(graphic, false);
  ```

  This is useful particularly if you want to create graphics that act as background images and labels should be placed on top of the background images. If the background images were obstacles, the label layout algorithm would try to avoid the area covered by the background images.

The overlap calculation of the default labeling model takes the bounding box of the obstacles into account. This is a good approximation for the most objects. For subclasses of `IlvLine`, `IlvPolyline`, and `IlvLinkImage`, it takes the precise polyline shape into account. However, the default labeling model does not work well with spline obstacles.

The Label Layout Base Class

The `IlvLabelLayout` class is the base class for all label layout algorithms. This class is an abstract class and cannot be used directly.

Subclasses of IlvLabelLayout

There are currently two subclasses:

- `IlvRandomLabelLayout` which randomizes the label positions for demonstration purpose.
- `IlvAnnealingLabelLayout` which does a useful label arrangement.

You can also create your own subclasses to implement other label layout algorithms.

Despite the fact that only subclasses of `IlvLabelLayout` are directly used to obtain the layouts, it is still necessary to learn about this class because it contains methods that are inherited (or overridden) by the subclasses. And, of course, you will need to understand it if you subclass it yourself.

Using IlvLabelLayout

The following sections describe the basic operations and generic parameters of `IlvLabelLayout`. For more details on the `IlvLabelLayout` class, see the *ILOG JViews Graph Layout Reference Manual*.

- *Instantiating and Attaching a Subclass of IlvLabelLayout*
- *Performing a Layout*
- *Performing a Recursive Layout on Nested Subgraphs*

- *The Label Layout Report*
- *Layout Events and Listeners*
- *Layout Parameters and Features in IlvLabelLayout*

Instantiating and Attaching a Subclass of IlvLabelLayout

The class `IlvLabelLayout` is an abstract class. It has no constructors. You will instantiate a subclass as shown in the following example:

```
IlvAnnealingLabelLayout layout = new IlvAnnealingLabelLayout();
```

In order to place labels, a manager needs to be attached to the layout instance. The following method, defined on the class `IlvLabelLayout`, allows you to specify the manager you want to lay out:

```
void attach(IlvManager manager)
```

For example:

```
...
IlvManager manager = new IlvManager();
/* Add obstacles and labels to the manager here */
layout.attach(manager);
```

The `attach` method does nothing if the specified manager is already attached. Otherwise, it first detaches the manager that is already attached, if any. You can obtain the attached manager using the method `getManager()`. If the manager is attached in this way, a default labeling model is created internally. The labeling model can be obtained by:

```
IlvLabelingModel labelingModel = layout.getLabelingModel();
```

Warning: *Notice that such an internally created model is not allowed to be attached to any other layout instance.*

After layout, when you no longer need the layout instance, you should call the `detach` method. If the `detach` method is not called, some objects may not be garbage-collected. This method also performs cleaning operations on the manager (properties that may have been added by the layout algorithm on the manager's objects are removed).

```
void detach()
```

Performing a Layout

The `performLayout` methods start the layout algorithm using the currently attached manager and the current settings for the layout parameters. The method returns a report object that contains information about the behavior of the label layout algorithm.

```
IlvLabelLayoutReport performLayout()
IlvLabelLayoutReport performLayout(boolean force, boolean redraw)
```

The first version of the method simply calls the second one with a `false` value for the first argument and a `true` value for the second argument. If the argument `force` is `false`, the layout algorithm first verifies whether it is necessary to perform the layout. It checks internal flags to see whether the manager or any of the parameters have been changed since the last time the layout was successfully performed. A "change" can be any of the following:

- Obstacles or labels were added or removed.
- Obstacles or labels were moved or reshaped.
- The value of a layout parameter was modified.
- The transformer changed while nonzoomable obstacles or labels were used.

Users often do not want the layout to be computed again if no changes occurred. If there were no changes, the method `performLayout` returns without performing the layout. Note that if the argument `force` is passed as `true`, the verification is skipped, and layout is performed even if no changes occurred.

The argument `redraw` determines whether a redraw of the manager is requested. This mechanism works in exactly the same way as in graph layout. For details, see *Redrawing the Grapher after Layout* on page 296.

The protected abstract method `layout(boolean redraw)` is then called. This means that control is passed to the subclasses that are implementing this method. The implementation computes the layout and moves the labels to new positions.

Performing a Recursive Layout on Nested Subgraphs

The examples and explanations above assume that you work with a flat manager. ILOG JViews allows you to nest a grapher as a node into another grapher. You can create a hierarchy of nested graphers (see Figure 6.2 on page 351; see also *Nested Managers and Nested Graphers* in the *ILOG JViews Graphics Framework User's Manual*). You can apply a recursive graph layout to this hierarchy by calling:

```
graphLayout.performLayout(true, true, true);
```

However, it usually makes no sense to apply a label layout alone to nested graphers. When labels are placed in a subgrapher, this will likely change the bounds of the subgrapher; hence the node positions in its parent grapher will no longer be up-to-date and a new graph layout will be necessary.

It makes sense to apply a label layout in combination with another graph layout to nested graphers.

1. First, the graph layout is applied to arrange the nodes and links nicely.

2. Then the label layout is applied to position the labels according to the node and link positions.

3. When this is finished for all subgraphers, then it can be done for the parent grapher.

To perform a graph layout and a label layout together, you can use the Multiple Layout class. This is a subclass of `IlvGraphLayout` that allows combining graph layouts with a label layout. The following sample shows how to apply a Tree Layout and an Annealing Label Layout in combination on a subgrapher.

```
IlvTreeLayout treeLayout = new IlvTreeLayout();
IlvAnnealingLabelLayout labelLayout = new IlvAnnealingLabelLayout();
IlvGraphLayout multipleLayout = new IlvMultipleLayout(treeLayout,
                                                      null,
                                                      labelLayout);
// Now set the parameters for tree layout and label layout ...
// Finally, perform a recursive layout that handles tree layout and label
// layout together
try {
    multipleLayout.performLayout(true, true, true);
} catch (IlvGraphLayoutException e) {
    ...
}
```

Thus, the label layout does not provide a separate mechanism for a recursive layout on submanagers. By incorporating the label layout into a multiple graph layout, you can use all the graph layout facilities that are available for nested graphs (see also *Laying Out Nested Graphs* on page 306 and *Recursive Layout* on page 265).

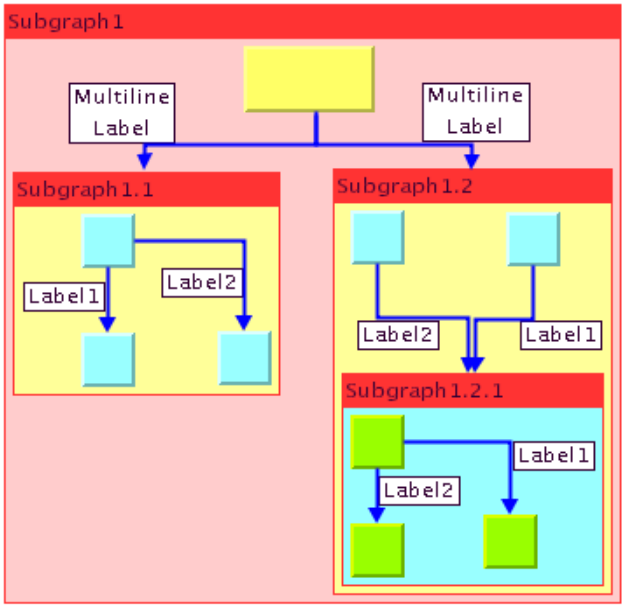


Figure 6.2 *Nested Subgraphers with Labels*

The Label Layout Report

The label layout report contains information about the particular behavior of a label layout algorithm. After the layout is completed, this information is available for reading from the label layout report. The information can also be obtained during layout by using a layout listener, as described in *Layout Events and Listeners* on page 352. The layout report is created automatically at the start of layout via the method `createLayoutReport` and is available as long as the manager is attached to the layout instance.

To read a layout report, all you need to do is store the layout report instance returned by the `performLayout` method and read the information, as shown in the following example:

```
...
IlvLabelLayoutReport layoutReport = labelLayout.performLayout();
if (layoutReport.getCode() == IlvLabelLayoutReport.LAYOUT_DONE)
     System.out.println("Label layout done.");
else
     System.out.println("Label layout not done, code = " +
                                          layoutReport.getCode());
```

The class `IlvLabelLayoutReport` stores the following information. The meaning is very similar to the information stored in an `IlvGraphLayoutReport` (see *Information Stored in a Layout Report* on page 293 for details):

- Code

  This field contains information about special, predefined cases that may have occurred during the layout. The possible values are the following:

  - `IlvLabelLayoutReport.LAYOUT_DONE`
  - `IlvLabelLayoutReport.STOPPED_AND_VALID`
  - `IlvLabelLayoutReport.STOPPED_AND_INVALID`
  - `IlvLabelLayoutReport.NOT_NEEDED`
  - `IlvLabelLayoutReport.NO_LABELS`
  - `IlvLabelLayoutReport.EXCEPTION_DURING_LAYOUT`

  To read the code, use the method:

  ```
  int getCode()
  ```

- Layout Time

  This field contains the total duration of the layout algorithm at the end of the layout. To read the time (in milliseconds), use the method:

  ```
  long getLayoutTime()
  ```

- Percentage of Completion

  This field contains an estimate of the percentage of the layout that has been completed. To access the percentage, use the method:

  ```
  int getPercentageComplete()
  ```

Layout Events and Listeners

The label layout framework provides the same event mechanism as the graph layout framework. The following events may occur:

- `LabelLayoutEvent` corresponds to `GraphLayoutEvent` (see *Layout Event Listeners* on page 294). You can install a listener to these events at the layout instance by:

  ```
  labelLayout.addLabelLayoutEventListener(listener);
  ```

  The listener must implement the `LabelLayoutEventListener` interface and receives events while the layout is running. A typical example is to check how much of the layout has already completed:

  ```
  class MyLabelLayoutListener
    implements LabelLayoutEventListener
  {
    public void layoutStepPerformed(LabelLayoutEvent event)
    {
      IlvLabelLayoutReport layoutReport = event.getLayoutReport();
      System.out.println("percentage of completion: " +
                         layoutReport.getPercentageComplete());
  ```

```
  }
}
```

- `LabelLayoutParameterEvent` corresponds to `GraphLayoutParameterEvent` (see *Parameter Event Listeners* on page 295). You can install a listener to these events at the layout instance by

  ```
  labelLayout.addLabelLayoutParameterEventListener(listener);
  ```

  The listener must implement the `LabelLayoutParameterEventListener` interface and receives events when layout parameters change. It also receives a special event at the end of a successful layout. For example:

  ```
  class MyLabelLayoutParameterListener
    implements LabelLayoutParameterEventListener
  {
    public void parametersUpToDate(LabelLayoutParameterEvent event)
    {
      if (!event.isParametersUpToDate())
        System.out.println("Any label layout parameter has changed.");
    }
  }
  ```

Layout Parameters and Features in IlvLabelLayout

The `IlvLabelLayout` class defines a number of generic features and parameters. These are a subset of the mechanism, methods, and parameters that are available for the `IlvGraphLayout` class. Therefore, we only list them here; for detailed explanations, refer to the particular section in *Layout Parameters and Features in IlvGraphLayout* on page 20 that describes the corresponding features for the `IlvGraphLayout` class.

Although the `IlvLabelLayout` class defines the generic parameters, it does not control how they are used by its subclasses. Each label layout algorithm (that is, each subclass of `IlvLabelLayout`) supports a subset of the generic features and determines the way in which it uses the generic parameters. When you create your own label layout algorithm by subclassing `IlvLabelLayout`, you decide whether to use the features and the way in which you are going to use them.

The `IlvLabelLayout` class defines the following generic features:

- *Allowed Time*
- *Percentage Completion Calculation*
- *Random Generator Seed Value*
- *Save Parameters to Named Properties*
- *Stop Immediately*
- *Use Default Parameters*

For details of all generic features, see *Layout Parameters and Features in IlvGraphLayout* on page 20.

The Annealing Label Layout

In this section, you will learn about the *Annealing Label Layout* algorithm from the ILOG JViews Label Layout module (class `IlvAnnealingLabelLayout` from the package `ilog.views.graphlayout.labellayout.annealing`).

Sample

Here are some sample drawings produced with the Annealing Label Layout:

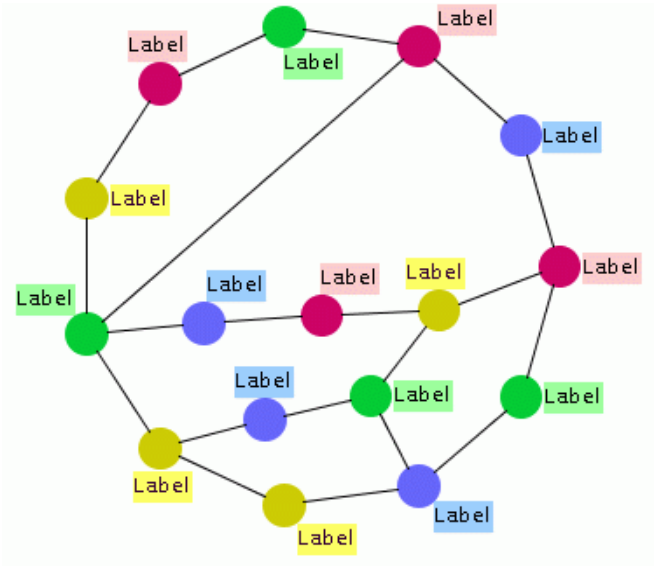


Figure 6.3 *Label Placement At Nodes with the Annealing Label Layout*

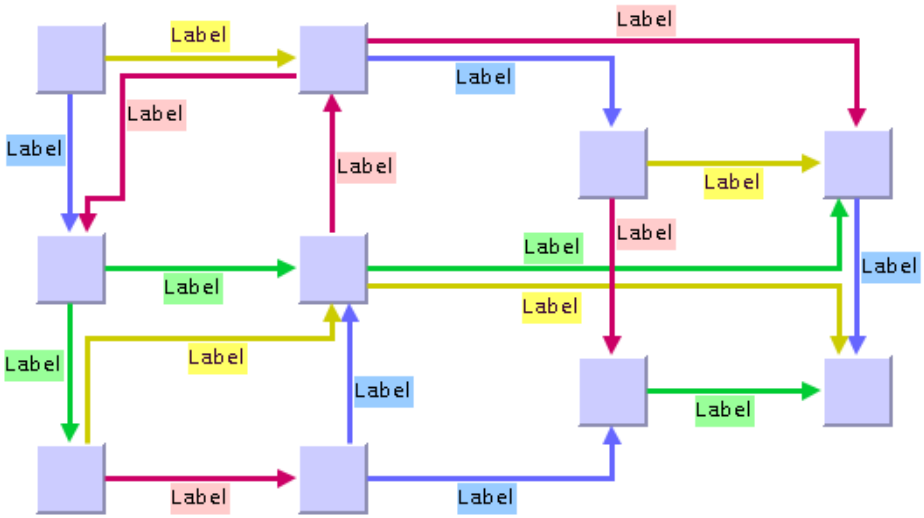


Figure 6.4 *Label Placement At Links with the Annealing Label Layout*

Figure 6.5 *Label Placement At Cities in a Map of Germany with Annealing Label Layout*

Features

- Places only labels. Does not move any obstacles around.
- Quality controlled, randomized iterative heuristic.
- Can place labels at points, rectangles, ellipses, and polylines.
- Can be used to place labels at nodes and links.
- Can be used to place labels at cities in a geographic map.
- Can place multiple labels at the same object (point, node, link, city, and so on).
- Tries to avoid overlaps among labels, and between labels and obstacles, by using the available free space.
- Provides several anchor and preference options.
- Easy extensible via subclassing of label descriptors.
- Efficient, scalable algorithm. Produces nice label placements even if the number of labels is large.

Limitations

- The Annealing Label Layout algorithm, as a randomized iterative heuristic, does not guarantee that labels are placed without overlaps whenever possible. However, it produces a high quality layout with a high probability of minimum overlap. The more iterations, the higher the probability of high quality.
- The algorithm is not able to create free space for labels by moving obstacles around. It is recommended that you perform a graph layout algorithm with large spacing parameters to create the necessary free space before placing the labels.
- While the algorithm is able to place labels at straight and polyline graphics, it is not able to place labels precisely at smooth curves such as spline graphics (`IlvSpline`) or spline links (`IlvSplineLinkImage`, `IlvOneSplineLinkImage`, `IlvDoubleSplineLinkImage`).
- The algorithm handles labels as upright rectangles. It is, for instance, not able to rotate a label to fit the flow of a link.

Brief Description of the Algorithm

The algorithm uses *simulated annealing*. This is a general, randomized optimization technique that simulates a thermodynamic particle system. Each label is moved to a new random position within the limits given by its label descriptor. The quality of the new position is calculated and compared to the quality of the old position. If the quality has not improved, the label is moved back to the old position. The amount of movement is

controlled by a conceptual temperature: when the system is hot, the labels can move long distances, producing potentially large global quality improvements. When the system cools down, the move distances become smaller and hence focus on local fine-tuning of the position.

Each label has its own label descriptor. The label descriptor describes the path on which the label can move. If a label must be placed at a city in a geographic map, then its label descriptor makes sure that the label is always placed close to the graphics that represent the city. If a label must be placed at a specific point, the `IlvAnnealingPointLabelDescriptor` can be used and describes an approximately elliptical path around the point. If a label must be placed at a polyline, the `IlvAnnealingPolylineLabelDescriptor` can be used and describes a boundary path at both sides of the polyline.

Code Sample

Here is a code sample using the `IlvAnnealingLabelLayout` class:

```
 ...
import ilog.views.*;
import ilog.views.graphlayout.labellayout.*;
import ilog.views.graphlayout.labellayout.annealing.*;
 ...
IlvManager manager = new IlvManager();
IlvManagerView view = new IlvManagerView(manager);

 ... /* Fill in here code that fills the manager with labels and obstacles */

IlvAnnealingLabelLayout layout = new IlvAnnealingLabelLayout();
layout.attach(manager);

/* Assume: label1 should be placed at the right side of rectangular node1 */
layout.setLabelDescriptor(
          label1,
          new IlvAnnealingPointLabelDescriptor(
                label1, node1, IlvAnnealingPointLabelDescriptor.RECTANGULAR,
                IlvDirection.Right));

... /* Fill in here code that sets descriptors for all other labels */

IlvLabelLayoutReport layoutReport = layout.performLayout();
if (layoutReport.getCode() == IlvLabelLayoutReport.LAYOUT_DONE)
     System.out.println("Layout done.");
else
     System.out.println("Layout not done, code = " +
                                    layoutReport.getCode());
```

Generic Features and Parameters

The `IlvAnnealingLabelLayout` class supports the following generic parameters defined in the `IlvLabelLayout` class:

- *Allowed Time*
- *Percentage Completion Calculation*
- *Random Generator Seed Value*
- *Save Parameters to Named Properties*
- *Stop Immediately*
- *Use Default Parameters*

The following comments describe the particular way in which these parameters are used by this subclass.

Allowed Time

The label layout algorithm stops if the allowed time setting has elapsed. (This feature works similarly as in `IlvGraphLayout`; see *Allowed Time* on page 22.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is:

- `IlvLabelLayoutReport.STOPPED_AND_VALID` if the labels were moved to some better (but not yet optimal) positions.
- `IlvLabelLayoutReport.STOPPED_AND_INVALID` if the time elapsed even before that.

Percentage Completion Calculation

The label layout algorithm calculates the estimated percentage of completion. This value can be obtained from the label layout report during the run of the layout. (For a detailed description of this feature, see *Percentage Completion Calculation* on page 31 and *Layout Events and Listeners* on page 352.)

Random Generator Seed Value

The Annealing Label Layout is a randomized heuristic. It uses a random number generator to control the movements. For the default behavior, the random generator is initialized using the current system clock. Therefore, different layouts are obtained if you perform the layout repeatedly on the same graph.

You can specify a particular value to be used as a seed value. For example, to specify the seed value `10`, call:

```
layout.setUseSeedValueForRandomGenerator(true);
layout.setSeedValueForRandomGenerator(10);
```

Save Parameters to Named Properties

The label layout algorithm can save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Using Named Properties to Save Layout Parameters* on page 382.)

Stop Immediately

The label layout algorithm stops after cleanup if the method `stopImmediately` is called. (This method works for the `IlvLabelLayout` class similarly to the corresponding method in the `IlvGraphLayout` class. For a description of this method in the `IlvGraphLayout` class, see *Stop Immediately* on page 35.) If the layout stops early in this way, the result code in the layout report is:

- `IlvLabelLayoutReport.STOPPED_AND_VALID` if the labels were moved to some better (but not yet optimal) positions.
- `IlvLabelLayoutReport.STOPPED_AND_INVALID` if the layout stopped even before that.

Use Default Parameters

After modifying any label layout parameter, you may want the layout algorithm to use the default values. You select the default values for all global parameters by:

```
layout.setUseDefaultParameters(true);
```

ILOG JViews keeps the previous settings when selecting the default values mode. You can switch back to your own settings by:

```
layout.setUseDefaultParameters(false);
```

This setting affects only the global layout parameters. The label descriptors have no default values, so parameters of the label descriptors do not change depending on this flag.

Label Descriptors

To specify where a label must be placed, you must set a label descriptor for each label. The algorithm places only those labels that have a label descriptor.

A label descriptor describes the locations that are allowed for the label. For instance, if you place a city name label in a geographic map, you want the label to be positioned close to the graphic objects that represent the city. Positions far away from the city are not reasonable for the label.

You need to allocate a new label descriptor for each label. To set the descriptor for one label, call:

```
layout.setLabelDescriptor(label, descriptor);
```

You can retrieve the current label descriptor by:

```
layout.getLabelDescriptor(label);
```

Subclasses of Label Descriptors

There are two predefined subclasses of label descriptors, the:

- *Point Label Descriptor*
- *Polyline Label Descriptor*

Depending on the parameters passed during the construction, these subclasses allow you to place a label:

- Close to a given point.
- Close to a specific rectangular or elliptic obstacle (such as a node).
- Along an imaginary polyline.
- Close to a polyline obstacle (for example, `IlvLine`, `IlvPolyline`).
- Close to a link.

You can also implement your own label descriptors by subclassing `IlvAnnealingLabelDescriptor`. This is explained in the section *For Experts: Implementing Your Own Label Descriptors* on page 373.

Point Label Descriptor

The `IlvAnnealingPointLabelDescriptor` can be used to place a label at a specific obstacle or point. This is known as the *point labeling problem*.

Positioning at an Obstacle

Here is an example of positioning a label at a specific obstacle:

```
layout.setLabelDescriptor(label,
    new IlvAnnealingPointLabelDescriptor(label, node,
          IlvAnnealingPointLabelDescriptor.ELLIPTIC, IlvDirection.Right));
```

This specification can be used if the label must be placed at a node that has an elliptical or circular shape. The label is placed in the free area around the node such that the border of the label just touches the border of the node (shown in Figure 6.6). The preferred position is the right side of the node, but this preferred position is used only if it does not create overlaps. If the node is moved or reshaped, the next call of label layout will update the position of the label automatically such that it follows the node.

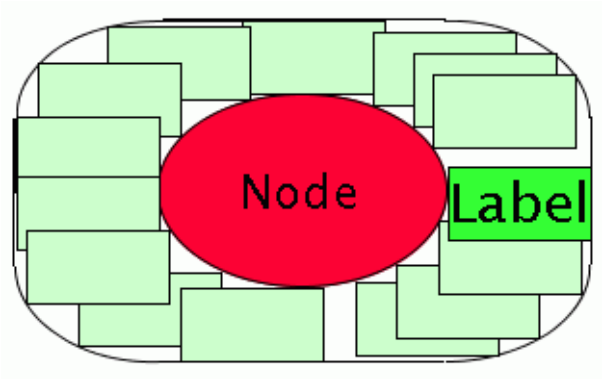


Figure 6.6 *Potential Label Positions Around a Node*

The example uses the following constructor:

```
IlvAnnealingPointLabelDescriptor(Object label,
        Object relatedObstacle, int shape, int preferredDirection)
```

The `relatedObstacle` is the obstacle that gets the label. The label is placed outside but close to this obstacle. The related obstacle can be a node of a graph, a city of a geographic map, a station of a railroad, and so on, whatever needs to have a label. The shape of the related obstacle should be either an ellipse, circle, or rectangle. You pass as the `shape` argument:

- `IlvAnnealingPointLabelDescriptor.ELLIPTIC` for ellipses or circles,
- `IlvAnnealingPointLabelDescriptor.RECTANGULAR` for rectangles.

If the real shape of the related obstacle is neither of these, then pass the shape that is the best approximation. For instance, if the obstacle is an `IlvRoundRectangle`, this can be considered as a rectangular shape.

The preferred position is a suggestion for the label layout algorithm where the label should preferably be placed. If the area at the preferred position is occupied, the label will be placed elsewhere. Options for the preferred position are:

- `IlvDirection.Left`
- `IlvDirection.Right`
- `IlvDirection.Top`
- `IlvDirection.Bottom`

Positioning at a Point

Here is an example of positioning a label at a specific point:

```
layout.setLabelDescriptor(label,
    new IlvAnnealingPointLabelDescriptor(label, null, new IlvPoint(25, 75),
          5f, 15f, IlvDirection.Right));
```

This specification can be used if the label must be placed close to a specific point (in this example at (25, 75)) independent of any obstacle. The label must be at least 5 coordinate units and at most 15 coordinate units away from the point (see Figure 6.7). The preferred position is at the right side of the point.

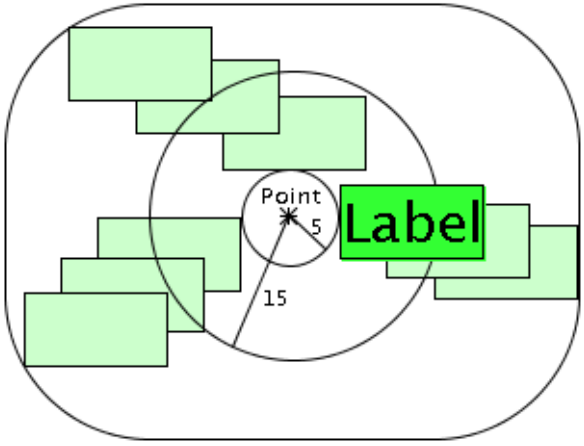


Figure 6.7 *Potential Label Positions Between 5 and 15 Units Away From a Point*

The example uses the following constructor:

```
IlvAnnealingPointLabelDescriptor(
    Object label, Object relatedObstacle, IlvPoint referencePoint,
     float minDist, float maxDist, int preferredDirection)
```

The label is placed close to the reference point. It does not take the actual position of the related obstacle into account. If the related obstacle is moved, the label does not follow the obstacle on the next call of layout, but stays at the reference point.

If a related obstacle is given, the label is not pushed away from the related obstacle. However, it is pushed away from all other obstacles to avoid overlaps. You can set the `relatedObstacle` parameter to `null` if the label is independent of all obstacles.

The parameters `minDist` and `maxDist` are the minimal and maximal distances from the reference point, measured from the border of the label. If you set the minimal and maximal distance to 0, the label will just touch the reference point. To keep the circular area around the reference point free, set the minimal distance accordingly. Most of the time you probably

want to keep the label close to the reference point; hence, set the minimal and maximal distances to the same value.

The preferred position indicates whether the label should be placed to the left, right, top, or bottom of the reference point. This is a suggestion for the labeling algorithm, as described for *Positioning at an Obstacle* on page 360.

Positioning on Multiple Criteria

The most powerful constructor combines all previously mentioned possibilities:

```
IlvAnnealingPointLabelDescriptor(
    Object label, Object relatedObstacle, IlvPoint referencePoint,
     int shape, float halfWidth, float halfHeight,
     float maxDistFromPath, float preferredDistFromPath,
     int preferredDirection)
```

If a related obstacle is given and the reference point is `null`, the label is placed close to the related obstacle. If a reference point is given, the label is placed close to the reference point independently from the related obstacle position.

The shape of the free area around the point can be rectangular or elliptic. The parameter `halfWidth` is the minimal distance of the label to the reference point in the horizontal direction. The parameter `halfHeight` is the minimal distance of the label to the reference point in the vertical direction. If the reference point is `null`, the parameters `halfWidth` and `halfHeight` are calculated from the bounding box of the related obstacle. The parameter `maxDelta` specifies the maximal additional distance allowed for the label (shown in Figure 6.8). The parameter `preferredDelta` specifies the preferred additional distance for the label. It should be between 0 and `maxDelta`.

The preferred position indicates whether the label should be placed to the left, right, top, or bottom of the reference point or related obstacle. This is a suggestion for the labeling algorithm, as described for *Positioning at an Obstacle* on page 360.

An alternative way to create a point label descriptor is to start with the empty descriptor:

```
descriptor = new IlvAnnealingPointLabelDescriptor();
```

Before the empty descriptor can be used, it must be filled with information on how the label should be placed. As a minimum, you need to specify a related obstacle or a reference point. For example:

```
descriptor.setRelatedObstacle(obstacle);
descriptor.setShape(IlvAnnealingPointLabelDescriptor.ELLIPTIC);
descriptor.setPreferredDirection(IlvDirection.Left);
```

At the end of all changes, the descriptor must be passed to the layout instance:

```
layout.setLabelDescriptor(label, descriptor);
```

Once the descriptor is passed to the layout instance, it should normally not be changed. If you need to change it later, you must pass it to the layout instance again.

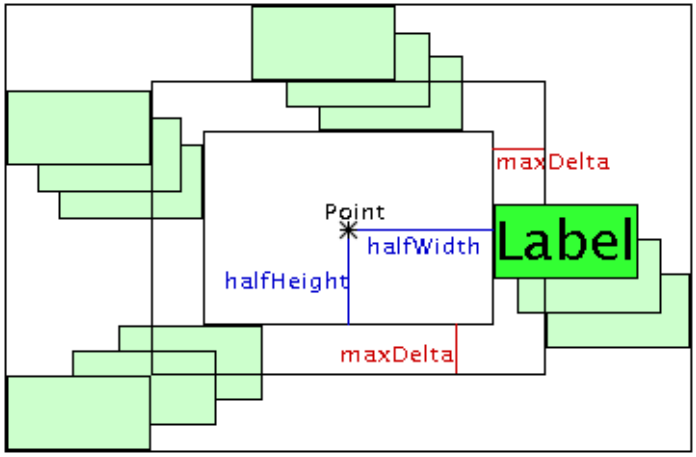


Figure 6.8 *Potential Label Positions With Rectangular Shape at a Point*

Polyline Label Descriptor

If you want to place labels at straight lines, polylines, or links, you should use the `IlvAnnealingPolylineLabelDescriptor`. The allowed area for labels at a polyline is different from the rectangular or elliptic area at a reference point. A polyline has two sides where the label can be placed along a path. This is known as the *polyline labeling problem*.

> ***Note:*** *The polyline label descriptor is not suitable for placing labels at splines or spline links. Because splines have a complex geometric shape, the automatic placement of labels at splines is currently not supported.*

Simple Positioning at a Link

This specification can be used if the label must be placed at a straight or polyline link. Here is an example:

```
layout.setLabelDescriptor(label,
    new IlvAnnealingPolylineLabelDescriptor(label, link,
          IlvAnnealingPolylineLabelDescriptor.CENTER));
```

The label is placed near the center of the link such that one border of the label just touches the link. If the link is moved or reshaped, the next label layout call will update the position of the label automatically to follow the link.

The example uses the following constructor:

```
IlvAnnealingPolylineLabelDescriptor(
    Object label, IlvLinkImage link, int anchor)
```

The options for the anchor are:

- `IlvAnnealingPolylineLabelDescriptor.CENTER` - place the label near the center of the link (that is, in the middle third of the link path).
- `IlvAnnealingPolylineLabelDescriptor.START` - place the label near the source node of the link (that is, in the first third of the link path).
- `IlvAnnealingPolylineLabelDescriptor.END` - place the label near the target node of the link (that is, in the last third of the link path).
- `IlvAnnealingPolylineLabelDescriptor.FREE` - place the label anywhere on the link.

Figure 6.9 *Anchors for Label Positions at a Link*

Simple Positioning at a Polyline Obstacle

This specification can be used if the label must be placed at a polyline obstacle (an instance of `IlvLine` or `IlvPolyline` in the default labeling model). The polyline obstacle does not need to be a link of a grapher.

Here is an example:

```
layout.setLabelDescriptor(label,
    new IlvAnnealingPolylineLabelDescriptor(label, polyline,
          IlvAnnealingPolylineLabelDescriptor.FREE,
          IlvDirection.Left, IlvDirection.TopLeft,
          IlvAnnealingPolylineLabelDescriptor.GLOBAL));
```

The label is placed anywhere at the left or top side of the polyline obstacle, with preference given to the left side.

The example uses the following constructor:

```
IlvAnnealingPolylineLabelDescriptor(
    Object label, Object relatedPolylineObstacle, int anchor,
    int preferredSide, int allowedSide, int sideAssociation)
```

Even though a polyline does not have a source node or a target node, the anchor specification can be used in the same way as for links (`CENTER`, `START`, `END`, and `FREE`). The first control point of the polyline is the start point. Labels with anchor `START` are placed closer to the first control point, and labels with anchor `END` are placed closer to the last control point of the polyline.

The *preferred* side is a suggestion for the label layout algorithm on where to place the label. If the area at the preferred side is occupied, the label is placed elsewhere. In contrast, the *allowed* side is a strict constraint: it is always obeyed, even if the entire area at the allowed side is occupied and the label must overlap the obstacles in this area.

Side Association

The orientation of the preferred and allowed sides depend on the *side association*. Options for the side association are listed below and illustrated in Figure 6.10:

- `IlvAnnealingPolylineLabelDescriptor.LOCAL`
- `IlvAnnealingPolylineLabelDescriptor.GLOBAL`

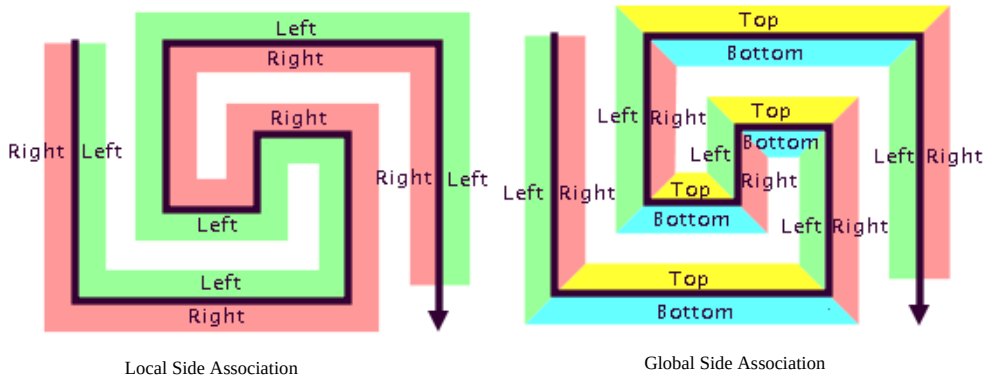


Figure 6.10 *Side Associations*

Local Side Association

If the side association is local, each polyline has two sides: left and right. The sides can be determined from the flow direction of the polyline from start point to end point. Consider yourself standing on the polyline looking in the direction where the polyline continues towards the end point, and then determine which is the left and which is the right side. Hence, the meaning of left and right in local side association is relative to the polyline. The options for the preferred and allowed side are in this case:

- `IlvDirection.Left`
- `IlvDirection.Right`

You can also specify the value 0 for the allowed side, which indicates that you do not want to restrict the side: both sides are allowed.

Global Side Association

If the side association is global, the side specification is independent from the flow direction of the polyline and more like a compass direction: north is top, south is bottom, west is left, and east is right. Here more options are possible: top, bottom, left, right, and all combinations of these. You specify the sides in the following way:

- `IlvDirection.Left`
- `IlvDirection.Right`
- `IlvDirection.Top`
- `IlvDirection.Bottom`

You can also use combinations of these, such as:

- `IlvDirection.Left | IlvDirection.Right` (left or right but not top or bottom).
- `IlvDirection.Left | IlvDirection.Top` (which is the same as `IlvDirection.TopLeft`, meaning the left or the top side).

You can specify the value 0 for the allowed side if all sides should be allowed.

Full Positioning at a Link

The most powerful constructor of a descriptor for a label that should be placed at a link is the following:

```
IlvAnnealingPolylineLabelDescriptor(
    Object label,
    Object link, Object source, Object target,
    int anchor,
    float maxDistFromLink, float preferredDistFromLink,
    int preferredSide, int allowedSide, int sideAssociation,
    float topOverlap, float bottomOverlap,
    float leftOverlap, float rightOverlap)
```

It combines all previously mentioned possibilities: you can specify the anchor, the side association, and the allowed and preferred sides.

You must pass the link as well as the source node and the target node of the link. The label does not need to touch the link. If you want to allow the label to be placed at a short distance from the link, then specify the maximal distance by `maxDistFromLink` and the preferred distance by `preferredDistFromLink`. Conversely, you may want to allow the link to partially overlap the label. You specify this by setting `topOverlap`, `bottomOverlap`, `leftOverlap`, or `rightOverlap` to a value larger than 0. This is illustrated in Figure 6.11.

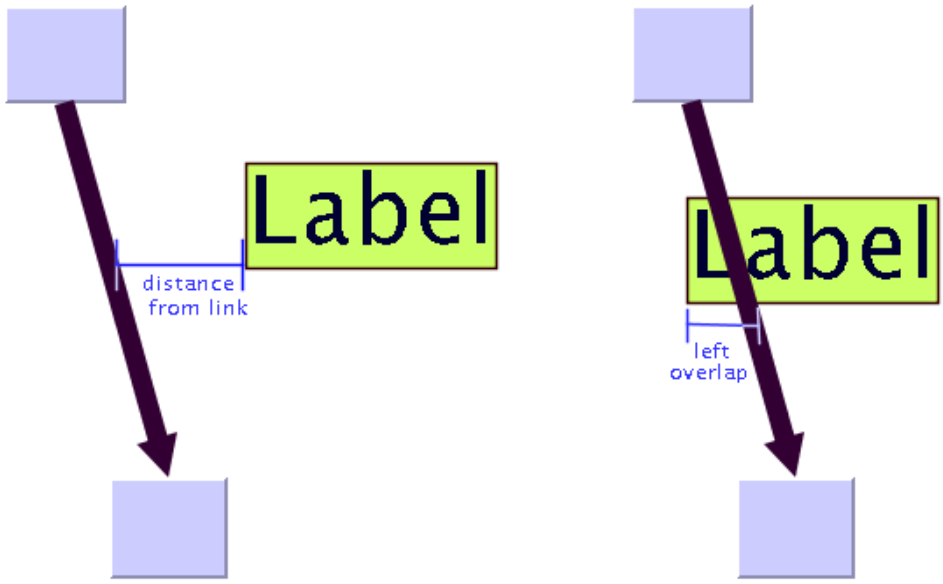


Figure 6.11 *Distance and Overlap at Link*

Full Positioning at a Polyline Obstacle

If the label should be placed at a polyline that is not a link, then the most powerful constructor is the following:

```
IlvAnnealingPolylineLabelDescriptor(
    Object label, Object relatedObstacle,
    IlvPoint[] referencePoints, float lineWidth,
    float minPercentageFromStart, float maxPercentageFromStart,
    float prefPercentageFromStart,
    float maxDistFromPath, float preferredDistFromPath,
    int preferredSide, int allowedSide, int sideAssociation,
    float topOverlap, float bottomOverlap,
    float leftOverlap, float rightOverlap)
```

It combines all previously mentioned possibilities. If the label should be placed at a polyline obstacle, then pass this as the related obstacle. If the label should be placed at an imaginary polyline, then pass the polyline with the `points` parameter and the width of the polyline with the `lineWidth` parameter. Instead of an anchor, you can pass the area where the label is placed by the minimal, maximal, and preferred percentage values relative to the polyline length. The minimal and maximal percentages are strictly obeyed, while the preferred percentage is only a recommendation for the layout.

- For instance, if you want to specify that the label can be placed anywhere but you prefer the center of the polyline, specify 0 and 100 for the minimal and maximal percentages and 50 for the preferred percentage. If there is not enough free space at the center, the label will be placed elsewhere.
- But if you want to specify that the label must be placed close to the center even if there is not enough space, then specify, for instance, 40 and 60 for the minimal and maximal percentages instead.

An alternative way to create a polyline label descriptor is to start with the empty descriptor:

```
descriptor = new IlvAnnealingPolylineLabelDescriptor();
```

Before the empty descriptor can be used, it must be filled with information on how the label should be placed. As a minimum, you need to specify a related obstacle or reference points and line width. For instance:

```
descriptor.setRelatedObstacle(polyline);
descriptor.setMinPercentageFromStart(10f);
descriptor.setMaxPercentageFromStart(90f);
descriptor.setPreferredPercentageFromStart(50f);
```

At the end of all changes, the descriptor must be passed to the layout instance:

```
layout.setLabelDescriptor(label, descriptor);
```

Once the descriptor is passed to the layout instance, it should normally not be changed. If you need to change it later, you must pass it to the layout instance again.

Specific Global Parameters

The following global parameters are specific to the IlvAnnealingLabelLayout class:

- *Label Offset*
- *Obstacle Offset*
- *Label Movement Policy*
- *Automatic Update*
- *Expert Parameters*

Label Offset

The label offset controls the desired minimal distance between two neighbored labels (see Figure 6.12 on page 370, left). To avoid labels being placed too close to each other, you can increase the label offset. This conceptually pushes the labels farther apart. However, depending on the available space, the minimal distance between labels cannot always be maintained.

Set the label offset as in the following statement:

```
layout.setLabelOffset(25f);
```

To obtain the label offset, call:

```
layout.getLabelOffset();
```

Obstacle Offset

The obstacle offset controls the desired minimal distance between a label and an unrelated obstacle. The obstacle offset is usually more important than the label offset because, if the obstacle offset is too small, the label may be placed so close to an unrelated obstacle that it incorrectly appears to be assigned to that obstacle (see Figure 6.12 on page 370, right: does, for example, the green label belong to the upper yellow node or to the green node?). Increasing the obstacle offset conceptually pushes the label away from the obstacle. However, depending on the available space, the minimal distance between label and obstacle cannot always be maintained.

Set the obstacle offset as in the following statement:

```
layout.setObstacleOffset(25f);
```

To obtain the obstacle offset, call:

```
layout.getObstacleOffset();
```

The obstacle offset should not be set to an unreasonably large value (such as `Float.MAX_VALUE`) because this can cause computational problems inside quadtree operations.

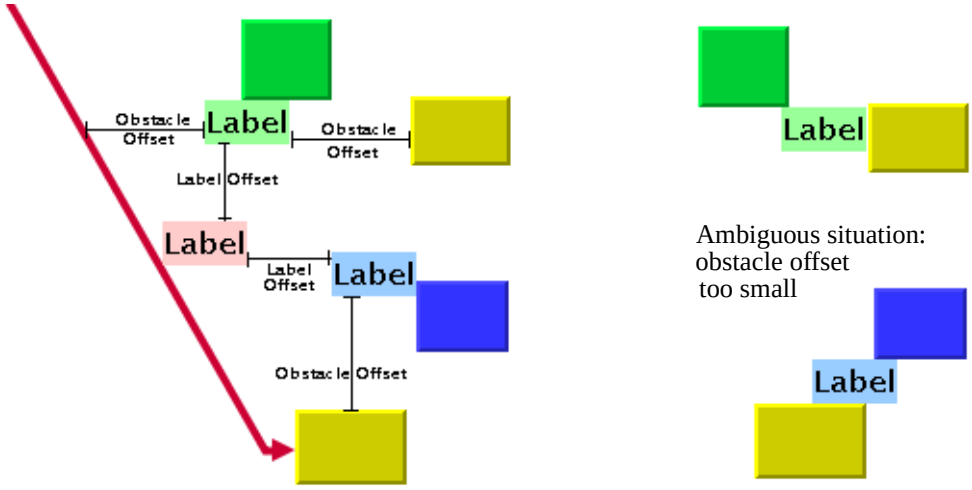


Figure 6.12 *Label and Obstacle Offsets*

The specified obstacle offset works globally for all labels. It is also possible to specify a smaller obstacle offset for specific label/obstacle pairs. You need to install an obstacle offset interface that returns the obstacle offset for a given pair:

```
layout.setObstacleOffsetInterface(new IlvObstacleOffsetInterface() {
    public float getOffset(IlvLabelingModel m, Object label, Object obstacle)
    {
        if (label instanceof IlvZoomableLabel &&
            obstacle instanceof IlvIcon)
            return 3f;
        else
            // use the global obstacle offset
            return Float.MAX_VALUE;
    }
});
```

The effective offset is the minimum of the value returned by the obstacle offset interface and the globally specified offset. Hence, the obstacle offset interface in the previous example means that labels of class `IlvZoomableLabel` can be placed up to 3 units near `IlvIcon` obstacles, while they are placed away from all other obstacles by at least the amount specified by `layout.setObstacleOffset(...)`.

Label Movement Policy

The label movement policy is an easy way to define which labels should be moved by the label layout algorithm.

For instance, the following code installs a label movement policy such that the layout moves only labels with a height larger than 100:

```
layout.setLabelMovementPolicy(new IlvLabelMovementPolicy() {
    public boolean allowMove(IlvLabelingModel labelingModel, Object label)
    {
        return (labelingModel.boundingBox(label).height > 100);
    }
});
```

Labels with smaller heights are not moved. However, they are also not completely ignored, because the layout tries to position the movable labels so that they do not overlap the immovable labels, and the label offset is respected between movable and immovable labels.

A more general useful example is a movement policy that prohibits moving labels that initially do not overlap anything. This predefined movement policy is available through the class `IlvOverlappingLabelMovementPolicy`. It can be used in applications that have their own label positioning mechanism and use the Annealing Label layout only as postprocessing step to improve the positions of overlapping labels. To install this policy, call:

```
layout.setLabelMovementPolicy(new IlvOverlappingLabelMovementPolicy());
```

Automatic Update

After layout, the labels are placed close to the related obstacle according to the label descriptor. For instance, a link label is placed close to its link. However, if you move the link interactively, the label normally stays at the old position, which may be far away from the link after the movement. The label loses the connection to the link, and a new layout is necessary.

Because it is too time-consuming to redo the layout after each single interactive move, the Annealing Layout algorithm has a feature that automatically updates the labels on geometric changes, that is, the label follows the link when the link moves. To enable this feature, call:

```
layout.setAutoUpdate(true);
```

You can check whether this feature is enabled by testing:

```
layout.isAutoUpdate();
```

If automatic update is enabled, the algorithm does not perform a full layout of all labels during each interactive change. It repositions only the label whose related obstacle has moved in one step. Thus it may produce more overlaps than a full layout. The automatic update mechanism is much faster, however, and hence better suitable for interactions.

> ***Note:*** *The automatic update feature works only if the labeling model provides* `LabelingModelEvents` *on each obstacle movement. The* `IlvDefaultLabelingModel` *provides these events. If you implement your own labeling model, you must provide these events in order to use the automatic update feature.*

Expert Parameters

To speed up the layout, the Annealing Label Layout algorithm uses a quadtree data structure. The quadtree allows a check for overlaps to be done very efficiently. The layout algorithm automatically detects from the graph model when the quadtree can be used. You can switch it off explicitly by calling:

```
layout.setUseQuadtree(false);
```

Normally it is not useful to switch the quadtree off because it slows down the positioning of the labels. However, if you implement your own labeling model, you may want to use this flag to verify that the labeling model is correct.

Simulated Annealing is an iterative mechanism. In each iteration step, all labels are tested for better positions. Usually the algorithm is able to automatically detect when to stop. The algorithm stops if:

- The maximal number of iterations is reached.
- After several iterations, no better position was found for any label.
- After several iterations, the quality did not improve by a given percentage.

In a few cases it may be necessary to limit the number of iterations, which can be done by the statement:

```
layout.setAllowedNumberOfIterations(100);
```

As a general hint, to obtain a reasonable layout, the allowed number of iterations should not be considerably lower than the number of labels.

Simulated Annealing stops if, after several iterations, no better position was found for any label. Because the search is randomized, this does not necessarily mean that the best position was already found; however, it indicates that finding the best position would require too much layout time. The number of ineffective iterations before stopping can be changed by:

```
layout.setMaxNumberOfFailIterations(maxNumber);
```

The default value is 20. If you set it to a higher value, the layout slows down but may find better positions for the labels. If you set it to a lower value, the layout stops sooner, but the label positions may be far from optimal.

In some cases, the algorithm improves the quality in each step, but the amount of improvement gets smaller in each step. In this situation, the previous fail-iteration criteria does not work well because there is an improvment in each step, but the amount of the improvement is so negligibly small that we want to stop. Therefore, it is also possible to require that the quality must improve in each step by a minimum percentage.

For example, to specify that the algorithm must improve over ten rounds by at least 2%, call:

```
layout.setNumberIterationsForMinImprovement(10);
layout.setMinImprovementPercentageToContinue(2);
```

By default, the layout stops if the quality did not improve by 0.5% over five iterations. If you set the required improvement percentage higher or the number of iterations lower, the layout stops sooner, but the label positions may be far from optimal. If you set the required percentage to 0%, this stop criterion is disabled and will no longer have any effect.

For Experts: Implementing Your Own Label Descriptors

The Annealing Label Layout is extensible. The point label descriptor and the polyline label descriptor are designed to cover the majority of the cases. In rare situations you may want to implement your own label descriptor by subclassing the base class `IlvAnnealingLabelDescriptor`. This section describes the necessary steps.

A label descriptor basically specifies the path where the top-left corner of a label can be placed. For simplification it considers the path rolled out such that the path has only one dimension. If the path is known, the precise label position can be specified by just one value: the path location. The Annealing Label Layout proposes different path locations during the layout; however, it does not know what the path looks like. The task of the label descriptor is to translate the path location into concrete (x, y) coordinates of the label.

As an example, we want to create a label descriptor that is able to place labels precisely at a triangular obstacle. We could use the point label descriptor as an approximation, but it does not place the labels precisely at a triangular shape.

Figure 6.13, top shows the path around the triangle (the dashed red and blue line). Below that you see the same path rolled out in one dimension. The Annealing Label Layout may ask the label descriptor to place the label at the position 1 to 8. For the Annealing Label Layout, these positions are just numbers between 0 and `maxPathLocation`. The task of the label descriptor is to translate these numbers into the correct positions as shown in the upper part of the figure.

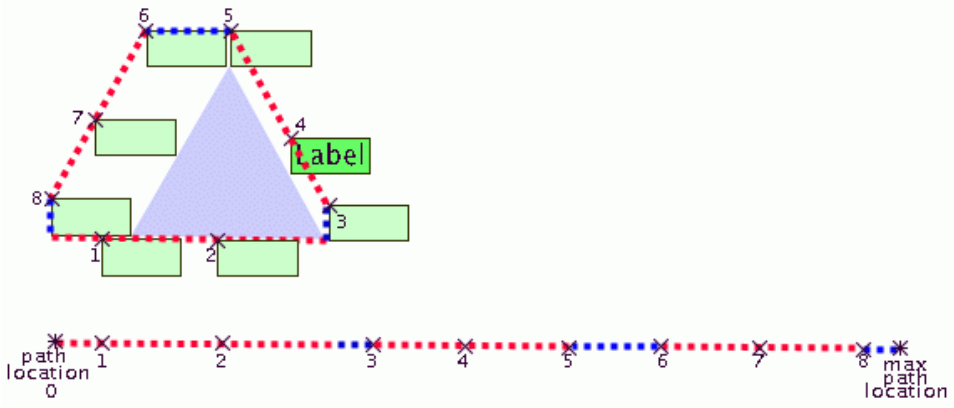


Figure 6.13 *Path Locations at a Triangle Label Descriptor*

The base class `IlvAnnealingLabelDescriptor` has two protected data members:

- `actPathLocation` is the current path location of the label.
- `maxPathLocation` is the maximal value of the path location.

To create a new label descriptor, you need to implement a method that initializes the path constructor at the beginning of layout. You should calculate the maximal path location `maxPathLocation` and initialize the `actPathLocation` here. The method is called only once during layout:

```
void initialize(IlvLabelingModel labelingModel)
```

In the example of Figure 6.13, the maximal path location for an equilateral triangle is:

```
3 * sidelength + 2 * labelwidth + 2  * labelheight
```

In each iteration step, the layout calls the method `setPosition` and provides an actual value of the path location. The method `setPosition` should store the value into `actPathLocation` and translate the path location into appropriate (x, y) coordinates. Then it should call the predefined method `updatePosition(x, y)` with these coordinates:

```
public void setPosition(double pathLocation, float distFromPath)
{
    float x, y;
    // make sure the position is between 0 and max
    while (pathLocation > maxPathLocation)
        pathLocation -= maxPathLocation;
    while (pathLocation < 0)
        pathLocation += maxPathLocation;
    // store the actual position
    actPathLocation = pathLocation;
    // translate the path location into (x, y)
    if (pathLocation < labelwidth + sidelength) {
        x = (float)pathLocation;
        y = triangleBottom;
    } else if (pathLocation < labelwidth + labelheight + sidelength) {
        x = labelwidth + sidelength;
        y = triangleBottom - (float)pathLocation + labelwidth + sidelength;
    } else if ... (other cases) ...
        ...
    // finally, update the internal data structures
    updatePosition(x, y);
}
```

The label may have a preferred position at the triangle. The Annealing Layout checks a location close to the preferred position from time to time. You should implement the following method to return the preferred path location:

```
double getPreferredPathLocation()
```

Furthermore, you should implement a strategy on how to come close to the preferred location. Towards the end of layout, the algorithm performs a sequence of steps that have the task of getting the label from the current position closer to the preferred position. For this, it calls the method:

```
setTowardsPreferredPosition(pathLocation, dist, i, maxI)
```

with `i` from `1` to `maxI`. You should implement the method such that in each step you calculate a path location closer to the preferred location. When `i` is `maxI`, it should be exactly at the preferred location. You can call `setPosition` to move the label to the preferred (x, y) position. For instance:

```
public void setTowardsPreferredPosition(
                double pathLocation, float dist, int i, int maxI)
{
    double offset = pathLocation - getPreferredPathLocation();
    double newLocation = pathLocation - i * offset / maxI;
    setPosition(newLocation, dist);
}
```

These methods take a distance as a parameter additional to the path location. This is the distance from the path. If the label must always be on the path, you can assume this distance is 0. Only if your label descriptor allows the label to have a variable offset to the path should you include the distance from the path in the calculations.

The Random Label Layout

In this section, you will learn about the *Random Label Layout* algorithm from the ILOG JViews Label Layout module (class `IlvRandomLabelLayout` from the package `ilog.views.graphlayout.labellayout.random`).

Sample

Here is a sample drawing produced with the Random Label Layout:

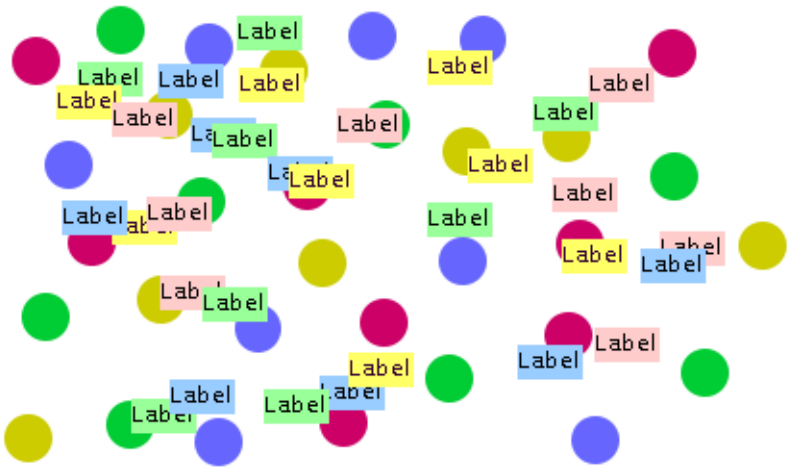


Figure 6.14 *Label Placement Produced with the Random Label Layout*

Although each label belongs to a circle of the same color, the random label placement does not show this. Instead it places the labels randomly, creating many overlaps. The Random Label Layout can be used to shuffle the labels arbitrary in a given area.

Features

- Mainly for demonstration purposes.
- Random placement of the labels inside a given region.

Brief Description of the Algorithm

The Random Label Layout algorithm is not really a useful layout algorithm. It simply places the labels at randomly computed positions inside a user-defined region. Nevertheless, the Random Label Layout algorithm may be useful for demonstration purposes.

Code Sample

Below is a code sample using the `IlvRandomLabelLayout` class:

```
 ...
import ilog.views.*;
import ilog.views.graphlayout.labellayout.*;
import ilog.views.graphlayout.labellayout.random.*;
 ...
IlvManager manager = new IlvManager();
IlvManagerView view = new IlvManagerView(manager);

 ... /*  Fill in here code that fills the manager with labels and obstacles */

IlvRandomLabelLayout layout = new IlvRandomLabelLayout();
layout.attach(manager);
layout.setLayoutRegion(new IlvRect(0, 0, 200, 200));
IlvLabelLayoutReport layoutReport = layout.performLayout();
if (layoutReport.getCode() == IlvLabelLayoutReport.LAYOUT_DONE)
     System.out.println("Layout done.");
else
     System.out.println("Layout not done, code = " +
                                    layoutReport.getCode());
```

Generic Features and Parameters

The `IlvRandomLabelLayout` class supports the following generic parameters defined in the `IlvLabelLayout` class:

- *Allowed Time*
- *Percentage Completion Calculation*
- *Random Generator Seed Value*
- *Save Parameters to Named Properties*
- *Stop Immediately*
- *Use Default Parameters*

The following comments describe the particular way in which these parameters are used by this subclass.

Allowed Time

The label layout algorithm stops if the allowed time setting has elapsed. (This feature works similarly as in `IlvGraphLayout`; see *Allowed Time* on page 22.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvLabelLayoutReport.STOPPED_AND_INVALID.`

Percentage Completion Calculation

The label layout algorithm calculates the estimated percentage of completion. This value can be obtained from the label layout report during the run of the layout. (For a detailed description of this features, see *Percentage Completion Calculation* on page 31 and *Layout Events and Listeners* on page 352.)

Random Generator Seed Value

The Random Label Layout uses a random number generator to compute the coordinates. For the default behavior, the random generator is initialized using the current system clock. Therefore, different layouts are obtained if you perform the layout repeatedly on the same graph.

You can specify a particular value to be used as a seed value. For example, to specify the seed value `10`, call:

```
layout.setUseSeedValueForRandomGenerator(true);
layout.setSeedValueForRandomGenerator(10);
```

Save Parameters to Named Properties

The label layout algorithm can save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Using Named Properties to Save Layout Parameters* on page 382.)

Stop Immediately

The label layout algorithm stops after cleanup if the method `stopImmediately` is called. (This method works for the `IlvLabelLayout` class similarly to the corresponding method in the `IlvGraphLayout` class. For a description of this method in the `IlvGraphLayout` class, see *Stop Immediately* on page 35.) If the layout stops early in this way, the result code in the layout report is `IlvLabelLayoutReport.STOPPED_AND_INVALID`.

Use Default Parameters

After modifying any label layout parameter, you may want the layout algorithm to use the default values. You select the default values for all parameters by:

```
layout.setUseDefaultParameters(true);
```

ILOG JViews keeps the previous settings when selecting the default values mode. You can switch between back to your own settings by:

```
layout.setUseDefaultParameters(false);
```

Specific Parameters

The following parameter is specific to the Ilv`RandomLabelLayout` class:

Layout Region

The Random Label Layout algorithm places the labels randomly in a specified region. You can set this region by:

```
setLayoutRegion(IlvRect region)
```

You can obtain the current region by:

```
getLayoutRegion()
```

If no layout region is specified, the default region (0, 0, 100, 100) is used.

Using Advanced Features

This section describes advanced features for using the Label Layout module of ILOG JViews.

Filtering Manager Layers

Inside an `IlvManager`, the graphic objects can be managed by layers. (See the `ilog.views.IlvManagerLayer` class). ILOG JViews allows you specify that only the labels and obstacles belonging to certain layers have to be taken into account when performing the layout. Use the following methods of the `IlvDefaultLabelingModel` class:

```
void addLayer(IlvManagerLayer layer)
boolean removeLayer(IlvManagerLayer layer)
boolean removeAllLayers()
```

To get an enumeration of the manager layers to be taken into account during the layout, use the method:

```
Enumeration getLayers()
```

To determine if a manager layer belongs to the layers to be taken into account during the layout, use the method:

```
boolean isLayerAdded(IlvManagerLayer layer)
```

If no layers have been specified or all the specified layers have been removed, all layers in the `IlvManager` are used. In this case, the `getLayers` method returns `null` and `isLayerAdded` returns `false` for any layer.

Nonzoomable Labels and Obstacles

Label layout algorithms have to deal with the geometry of the labels and obstacles. The labels and obstacles of an ILOG JViews `IlvManager` can be any graphic objects, that is, any subclass of `IlvGraphic`. Their position and size are given by their

`boundingBox(IlvTransformer t)` method and usually depend on the transformer used for their display. Therefore, when an `IlvManager` has to be laid out, the geometry of the manager must be considered for a given value of the transformer.

The most natural transformer value that could be chosen is the "identity" transformer. An identity transformer has no translation, zoom, or rotation factors. In terms of ILOG JViews, this would mean that the geometry of the `IlvManager` would be considered in the manager coordinates, not in the manager view coordinates (transformed coordinates). However, the special case of *nonzoomable* graphic objects must be taken into account. For this case, the idea of simply using the geometry of the grapher in manager coordinates is not pertinent.

A Special Case: Nonzoomable Graphic Objects

A graphic object is said to be *zoomable* if its bounding box follows the zoom level. Otherwise, the object is *nonzoomable*. For instance, the `IlvLabel` objects are nonzoomable while the `IlvZoomableLabel` objects are zoomable. (To determine whether a graphic object is zoomable, use its `boolean zoomable()` method or check its documentation.)

If all the graphic objects of an `IlvManager` are zoomable, a layout obtained on the basis of the graph geometry in manager coordinates will appear the same for any value of the transformer used for the display. Simply speaking, the drawing of the manager will just be zoomed, or translated.

When at least one nonzoomable graphic object is used as a label or obstacle, the geometry in manager coordinates can no longer be used. When drawn with different transformer values (for instance, at different zoom levels), the same `IlvManager` can look very different. In this case, you cannot use multiple manager views because only one of them can look correct. All other views will look wrong.

Coordinates Modes

The default labeling model provides several coordinates modes. The coordinates mode can be specified on the default labeling model and on the layout instance. The coordinates mode of the layout instance is used during layout, while the coordinates mode specified for the default labeling model is used on operations of the labeling model when layout is not currently performed.

To specify the coordinates mode, use the following method. It is available in the class `IlvDefaultLabelingModel` and in the class `IlvLabelLayout`:

```
void setCoordinatesMode(int mode)
```

Valid options of the coordinates mode are:

- `IlvLabelLayout.MANAGER_COORDINATES` - The geometry of obstacles and labels is computed using the coordinate space of the manager without applying any transformation. This mode is suitable if the manager does not contain any nonzoomable labels.

- `IlvLabelLayout.VIEW_COORDINATES` - The geometry of obstacles and labels is computed in the coordinate space of the reference manager view. This mode is suitable if the manager contains nonzoomable objects. The layout will be correct with respect to the reference view but not correct with respect to any other view. Dimensional layout parameters (such as the label offset) are specified in the coordinate space of the reference view.
- `IlvLabelLayout.INVERSE_VIEW_COORDINATES` - This is the default. The geometry of the graph is computed using the coordinate space of the reference manager view and then applying the inverse transformation. This simulates the manager coordinate space. This mode is also suitable if the manager contains nonzoomable objects. The layout will be correct only with respect to the reference view. Dimensional layout parameters (such as the label offset) are specified in the coordinate space of the manager.

For instance, to make sure that manager coordinates are used during layout, call:

```
layout.setCoordinatesMode(IlvLabelLayout.MANAGER_COORDINATES);
```

This does not change the coordinates mode of the labeling model until layout is started. Most of the time, however, it is recommended that you use the same coordinates mode for the default labeling model and the layout instance, so you call it twice:

```
defaultLabelingModel.setCoordinatesMode(IlvLabelLayout.MANAGER_COORDINATES);
layout.setCoordinatesMode(IlvLabelLayout.MANAGER_COORDINATES);
```

Reference View and Reference Transformer

To specify the reference view, use the method:

```
void setReferenceView(IlvManagerView view)
```

If no reference view is explicitly specified, the first manager view is used.

The default labeling model needs only to know the transformer of the reference view. Alternatively, you can specify the reference transformer directly. Use the method:

```
void setReferenceTransformer(IlvTransformer transformer)
```

The mechanism of how the reference view and reference transformer are used is similar to that in the `IlvGrapherAdapter` class of the Graph Layout module. For details, see the sections *Reference Transformers* on page 324 and *Reference Views* on page 325.

> ***Note:*** *The reference transformer is used only if the coordinates mode is* `IlvLabelLayout.VIEW_COORDINATES` *or* `IlvLabelLayout.INVERSE_VIEW_COORDINATES`*. If you change the reference transformer, the layout is no longer up-to-date. A subsequent layout is necessary. Hence, changing the zoom level of the reference view also renders the layout out-of-date.*

Using Named Properties to Save Layout Parameters

The ILOG JViews Graph Layout module offers the facility to convert graph layout parameters into named properties of the `IlvGrapher`. The same facility is available to convert label layout parameters into named properties of the `IlvManager`. Named properties can be stored in `.ivl` files.

The following method indicates whether a label layout class supports this mechanism:

```
supportsSaveParametersToNamedProperties()
```

It returns `true` if the layout class can transfer the parameter settings to named properties.

Saving Layout Parameters to .ivl Files

The following example shows how to save the `IlvManager`, including all label layout parameter settings, into an `.ivl` file.

```
IlvDefaultLabelingModel labelingModel =
   (IlvDefaultLabelingModel)labelLayout.getLabelingModel();

// transfer the layout parameters to named properties
if (labelLayout.supportsSaveParametersToNamedProperties())
    labelingModel.saveParametersToNamedProperties(labelLayout, false);
// save the attached manager with the named layout properties to file
labelingModel.getManager().write("abcd.ivl");

// remove the named layout properties because they are no longer needed
labelingModel.removeParametersFromNamedProperties();
```

The mechanism is the same as in the Graph Layout module. See *Saving Layout Parameters to .ivl Files* on page 315.

Loading Layout Parameters from .ivl Files

The following example shows how to load and recover the parameters of the label layout when the layout settings are stored in an `.ivl` file:

```
// Read the IVL file. This reads all named properties as well.
manager.read("abcd.ivl");

IlvDefaultLabelingModel labelingModel =
   (IlvDefaultLabelingModel)labelLayout.getLabelingModel();

// Transfer the parameter settings from the named properties to the layout.
// At this time point, the manager must be attached to the label layout
labelingModel.loadParametersFromNamedProperties(labelLayout);

// just to be sure that no named layout properties remain in the memory
labelingModel.removeParametersFromNamedProperties();
```

The mechanism is the same as in the Graph Layout Module. See *Loading Layout Parameters from .ivl Files* on page 318.

Releasing Resources Used During the Layout

Various objects need to be created during the layout process. Most commonly, these are layout instances (subclasses of `IlvLabelLayout`) and labeling models (subclasses of `IlvLabelingModel`). Some of the layout parameters are internally stored as property objects attached to the labeling model, to the manager, or to its labels and obstacles.

Methods to Release Resources

`IlvLabelLayout` provides four methods that help to release all these objects once they are no longer needed (such obsolete objects can be garbage collected and memory leaks (in the Java sense) avoided):

```
void detach()

void cleanLabelingModel(IlvLabelingModel labelingModel)

void cleanLabel(IlvLabelingModel labelingModel, Object label)

void cleanObstacle(IlvLabelingModel labelingModel,
                   Object obstacle)
```

The method `cleanLabelingModel` releases property objects that the layout instances stored globally for the labeling model or for the manager. The methods `cleanLabel` and `cleanObstacle` release property objects stored for the label or obstacle passed as an argument. These three methods are automatically called when `detach()` is called.

Rules for Releasing Resources

The following rules must be respected:

- When a layout instance instantiated by your code is no longer useful, call the method `IlvLabelLayout.detach()` on it to ensure that no manager or labeling model is still attached to it. Note that you can freely reuse a layout instance once the previously attached model has been detached.
- Layout parameters or stored property objects that are specific to a label or an obstacle are cleaned when calling `detach()`. This cleaning is done only for labels and obstacles that are still in the manager when the `detach()` method is called.
- If labels or obstacles need to be removed while a layout instance is attached (that is, before `detach()` is called), and you use a default labeling model (class `IlvDefaultLabelingModel`), the cleaning of properties for the removed labels and obstacles is done automatically. The default labeling model fires labeling model events to the layout when objects are removed from the manager. The layout listens to these events and calls `cleanLabel` and `cleanObstacle` automatically.

  Notice that this also holds when you attach the layout directly to the manager by calling `attach(IlvManager manager)`, because in this case, the default labeling model is always created and used internally.

- If labels or obstacles need to be removed while a layout instance is attached and you use your own subclass of `IlvLabelingModel` (see *Defining Your Own Labeling Model* on page 384), the cleaning is done automatically only if your labeling model fires labeling model events whenever an object gets removed. If no events are fired, you must call the methods `cleanLabel` or `cleanObstacle` yourself after removing any label or obstacle from the manager to perform the cleaning for it.
- When a labeling model instantiated by your code is no longer useful, call the method `dispose()` on it to ensure that the resources it has used are released. Note that a labeling model must not be used once it has been disposed of.

Summary

As a quick summary:

- If you attach a manager directly by `attach(manager)`, you need only make sure to call `detach()` when the layout is no longer used. Everything else is done automatically.
- If you allocate a default labeling model `defaultLabelingModel` yourself and call `attach(defaultLabelingModel)`, you need to call `detach()` on the layout as above, and eventually `defaultLabelingModel.dispose()` when the model is no longer used. Everything else is done automatically.
- If you attach a labeling model that is not an `IlvDefaultLabelingModel`, you need to call `detach()` on the layout, `dispose()` on the labeling model, and eventually `cleanLabel` or `cleanObstacle` whenever a label or obstacle gets removed while the layout is attached.

Defining Your Own Labeling Model

> ***Note:*** *Before reading this section, you should be familiar with the IlvLabelingModel class (see The Labeling Model on page 346).*

It is sometimes necessary to add label layout features to an existing application. If the application already uses the ILOG JViews manager (`IlvManager`) to manipulate and display labels and obstacles, using the label layout algorithms provided in ILOG JViews is a straightforward process. You can use and adapt the default labeling model (`IlvDefaultLabelingModel`). Even if the application uses data structures that are independent from the ILOG JViews data structures, it is possible to apply a label layout algorithm provided in ILOG JViews. You have to define your own labeling model by creating a subclass of `IlvLabelingModel`. The process is described in the following sections:

- *The IlvLabelingModel Class*
- *Subclassing the Default Labeling Model*

- *Creating a New Labeling Model*

The IlvLabelingModel Class

The methods defined in the `IlvLabelingModel` class can be divided into several categories: those that provide information on the structure of the labels and obstacles, on their geometry, on their overlap penalty, and notification of changes in the manager. They are described in the following sections:

- *Information on the Structure of Labels and Obstacles*
- *Information on the Geometry of Labels and Obstacles*
- *Overlap Calculation*
- *Notification of Changes*
- *Storing and Retrieving Object Properties*

Information on the Structure of Labels and Obstacles

The following methods of the `IlvLabelingModel` class allow the layout algorithms to retrieve information on the labels:

`boolean isLabel(Object obj)`	Decides whether an object is a label.
`Enumeration getLabels()`	Enumerates all existing labels.
`int getLabelsCount()`	Returns the number of labels that exist.

The first method decides whether an object is a label. The second method enumerates all existing labels. The third method returns the number of labels that exist.

The following methods allow the layout algorithms to retrieve information on the obstacles in a similar way:

```
boolean isObstacle(Object obj)
boolean isPolylineObstacle(Object obj)
Enumeration getObstacles()
int getObstaclesCount()
```

For optimization purposes, the labeling model distinguishes between normal obstacles and polyline obstacles. While normal obstacles cover the major part of their bounding box, polyline obstacles have a line width and range over intermediate points; thus they cover only a small part of their bounding box.

Since a polyline obstacle is an obstacle, both `isObstacle` and `isPolylineObstacle` return `true` for a polyline obstacle.

Information on the Geometry of Labels and Obstacles

For labels and obstacles, the label layout can retrieve the bounding box with the method:

```
IlvRect boundingBox(Object labelOrObstacle)
```

For the special polyline obstacles, the label layout can retrieve the precise shape of the polyline with the methods:

```
float getPolylineWidth(Object polylineObstacle)
```

```
IlvPoint[] getPolylinePoints(Object polylineObstacle)
```

The following method moves a label to the new position.

```
void moveLabel(Object label, float x, float y, boolean redraw)
```

Overlap Calculation

A good label layout avoids overlaps. Thus, the calculation of overlap values is an important step of the algorithm. The speed of the layout algorithm depends crucially on the speed of the overlap calculation. The labeling model provides the following methods for overlap calculations:

```
double getLabelOverlap(
  Object label1, IlvRect bbox1,
  Object label2, IlvRect bbox2,
  float minDist)
```

Calculates the overlap between two labels.

```
double getObstacleOverlap(
  Object label, IlvRect labelBBox,
  Object obstacle, IlvRect obstacleBBox,
  float minDist)
```

Calculates the overlap between a label and a normal obstacle.

```
double getPolylineObstacleOverlap(
  Object label, IlvRect labelBBox,
  Object polylineObstacle,
  IlvPoint[] pts, float lineWidth, float minDist)
```

Calculates the overlap between a label and a polyline obstacle.

These methods return a positive penalty value that indicates how much the objects overlap.

- The returned value is 0 if the objects do not overlap.
- A smaller overlap value designates less overlap than a larger overlap value. The actual number is arbitrary and depends on the implementer of the labeling model. For example, if all objects are rectangles, then it could be the size of the overlapping area of the rectangles.

Typically the overlap value is calculated before the label is moved. It is calculated for a speculative label position, not for the real label position. Hence, the speculative bounding box of the label is passed as an argument. Similarly, the bounding box (or polyline shape) of the obstacle is passed as an argument as well. The meaning of the returned value is the

overlap penalty if the label were placed at its passed bounding box, and the obstacle were placed at its passed bounding box.

If the overlap methods return a positive nonzero penalty only when the speculative bounding boxes overlap, then the following method can return `true`:

```
boolean isBoundingBoxDependent()
```

This method exists mainly for quadtree optimization purposes.

Notification of Changes

The following methods of the `IlvLabelingModel` class allow a layout algorithm to be notified of changes in the data structures:

```
void addLabelingModelListener(LabelingModelListener listener)

void removeLabelingModelListener(LabelingModelListener listener)

void fireLabelingModelEvent(Object obstacleOrLabel,
                            int eventType,
                              boolean adjusting)

void fireLabelingModelEvent(LabelingModelEvent event)
```

A “change” can be a structural change (that is, a label or obstacle was added or removed) or a geometrical change (that is, a label or obstacle was moved or reshaped). The event type is typically a bitwise-Or of the bit masks defined in the class `LabelingModelEvent`. For instance, when a label was removed, an event with type `(STRUCTURE_CHANGED | LABEL_REMOVED)` is fired and the removed label is stored in the event. The labeling model event listener mechanism provides a means to keep the layout algorithms informed of these changes. When the layout algorithm is restarted on the same graph, it is able to detect whether the data structures have changed since the last time the layout was successfully performed. If necessary the layout can be performed again. If there was no change, the layout algorithm can avoid unnecessary work by not performing the layout.

The labeling model event listener is defined by the `LabelingModelListener` interface. To receive the events (that is, instances of the `LabelingModelEvent` class), a class must implement the `LabelingModelListener` interface and must register itself using the `addLabelingModelListener` method of the `IlvLabelingModel` class.

> ***Note:*** *The label layout algorithms register themselves as listeners to the labeling model (via the functionality in* `IlvLabelLayout`*). Therefore, there is usually no need to manipulate these listeners directly.*

Storing and Retrieving Object Properties

The following methods of the `IlvLabelingModel` class allow a layout algorithm to store data objects for each label or obstacle:

```
void setProperty(
    Object labelOrObstacle, String key, Object value)
```

```
Object getProperty(Object labelOrObstacle, String key)
void setProperty(String key, Object value)
Object getProperty(String key)
```

The layout algorithm may need to associate a set of properties with the labels or obstacles, or global properties. Properties are a set of `key-value` pairs, where the `key` is a `String` object and the `value` can be any kind of information value.

Subclassing the Default Labeling Model

The default labeling model `IlvDefaultLabelingModel` is a subclass of `IlvLabelingModel`. It has certain properties that may not be suitable for your application:

- Only objects of type `IlvLabel` and `IlvZoomableLabel` are considered labels.
- Only objects of type `IlvLine`, `IlvPolyline`, and `IlvLinkImage` are considered polyline obstacles.
- All other objects are considered rectangular obstacles.

It is easy to change these properties by subclassing `IlvDefaultLabelingModel`. For instance, if you use only labels of class `MySpecialLabel`, you can override the method `isDefaultLabelClass`:

```
public boolean isDefaultLabelClass(Object obj)
{
    return (obj instanceof MySpecialLabel);
}
```

If objects should be completely ignored, you should make sure they are not considered as labels or obstacles. To avoid considering objects of class `IgnorableGraphic` as obstacles, you can override the method `isObstacle`:

```
public boolean isDefaultObstacleClass(Object obj)
{
    return super.isDefaultObstacleClass(obj) &&
           !(obj instanceof IgnorableGraphic);
}
```

> ***Note:*** *Instead of overriding* `isDefaultLabelClass` *and* `isDefaultObstacleClass`*, you could also specify which objects are labels and obstacles by using the methods* `setLabel` *and* `setObstacle`*, as illustrated in The Labeling Model on page 346.*

Some obstacles do not have a rectangular shape. For simplicity and speed, the overlap value is based on the bounding box of normal obstacles. Hence the default labeling model may compute overlaps in situations where in fact there are no overlaps, because the bounding box is usually a little larger than the area that is really covered by the obstacle. You can correct

this by overriding the method `getOverlapValue` by implementing a more precise (but also more complex) overlap test:

```
public double getObstacleOverlap(
    Object label, IlvRect labelBBox,
    Object obstacle, IlvRect obstacleBBox,
    float minDist)
  {
    ... complex calulation of the overlap considering the precise shape of
    ... the obstacle. If the label is closer than minDist to the obstacle,
    ... it should be considered as overlap
    ...
    if (hasOverlap)
        return a value proportional to the overlap;
    else
        return 0.0;
   }
```

Creating a New Labeling Model

The default labeling model is suitable only if the underlying data structure is an `IlvManager`. The case may arise where an application uses its own classes and where, for some reason, you do not want to replace these classes with ILOG JViews classes such as `IlvManager` and `IlvLabel`. Here you cannot use the default labeling model. To enable the label layout algorithms to work with these data structures, a custom labeling model (that is, a subclass of `IlvLabelingModel`) must be written.

The custom labeling model must implement all the *abstract* methods of the `IlvLabelingModel` class. The nonabstract methods of this class have a default implementation that is functional. However, they may not be optimal because they do not take advantage of the characteristics of the underlying graph implementation. In this case, they can be overridden as well. The efficiency of the layout algorithm depends directly on the efficiency of the implementation of the labeling model and the underlying data structure.

The following is the minimal set of methods that must be implemented:

```
abstract Enumeration getLabels()

abstract boolean isLabel(Object obj)

abstract void moveLabel(Object label, float x, float y,
    boolean redraw)

abstract double getLabelOverlap(
    Object label1, IlvRect bbox1,
    Object label2, IlvRect bbox2, float minDist)

abstract Enumeration getObstacles()

abstract boolean isObstacle(Object obj)
```

```
abstract double getObstacleOverlap(
    Object label, IlvRect labelBBox,
    Object obstacle, IlvRect obstacleBBox, float minDist)

abstract IlvRect boundingBox(Object labelOrObstacle)
```

These methods are described in *The IlvLabelingModel Class* on page 385.

C H A P T E R

7

Using Graph Layout Beans

The graph layout algorithm classes provided by ILOG JViews Graph Layout module are fully compliant with the JavaBeans™ standard. This allows you to create an ILOG JViews application or applet from the visual programming environment of your favorite Integrated Development Environment (IDE).

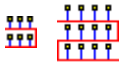
This chapter shows you how to use ILOG JViews Beans and the Graph Layout Beans when creating an applet within an IDE. The following topics are covered:

- *Graph Layout Classes Available as Beans*
- *Creating a Simple Applet Using ILOG JViews Graph Layout Beans*

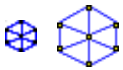
Before you can create an ILOG JViews application or applet within an Integrated Development Environment (IDE), you must install the ILOG JViews Beans into your IDE. Complete installation instructions are given in the *ILOG JViews Graphics Framework User's Manual*.

Graph Layout Classes Available as Beans

The following classes are provided as ILOG JViews Graph Layout Beans:

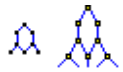


`IlvBusLayout` displays bus network topologies, that is, a set of nodes connected to a bus node.

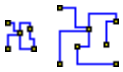


`IlvCircularLayout` displays graphs representing interconnected ring and/or star network topologies.

`IlvGridLayout` arranges disconnected nodes in rows or in columns or on a grid.



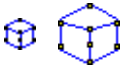
`IlvHierarchicalLayout` arranges nodes in horizontal or vertical levels such that the links flow in a uniform direction.



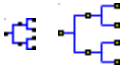
`IlvLinkLayout` reshapes the links of a graph without moving the nodes.

`IlvRandomLayout` moves the nodes of the graphs at randomly computed positions inside a user-defined area.

`IlvSpringEmbedderLayout` can be used to lay out any type of graph.



`IlvTopologicalMeshLayout` can be used to lay out cyclic graphs.



`IlvTreeLayout` arranges the nodes of a tree horizontally or vertically, starting from the root of the tree.

`IlvUniformLengthEdgesLayout` can be used to lay out any type of graph and allows you to specify a preferred length of the links.

`IlvJGraphLayoutProgressBar` a Swing JProgressBar toolbar that automatically displays the progress of the layout process.

The Beans listed in this section are classes of the ILOG JViews Graph Layout module. The details of these classes are explained throughout this manual and in the *ILOG JViews Graph Layout Reference Manual*.

Creating a Simple Applet Using ILOG JViews Graph Layout Beans

In this section, you are going to create an ILOG JViews applet using ILOG JViews Graphics Framework Beans. You are then going to add Graph Layout features using the Graph Layout Beans. The applet is a simple JFC/Swing applet that displays a graph and provides Graph Layout capabilities. No coding will be necessary.

Figure 7.1 shows the application panel that you will have created if you follow this short tutorial example.

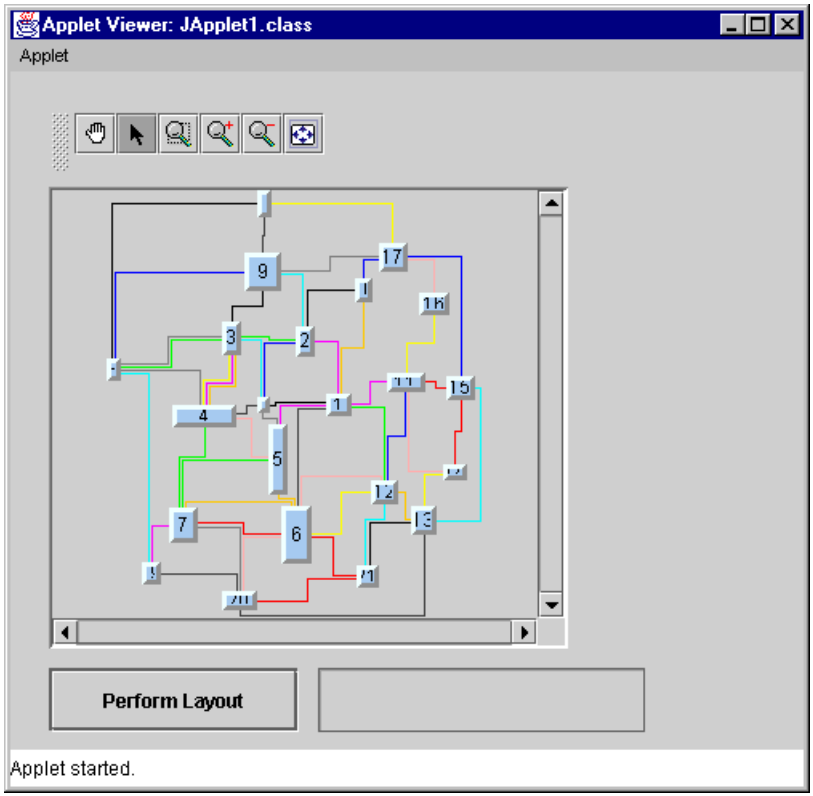


Figure 7.1 *Final JFC/Swing Applet*

The example in this section shows how to create an ILOG JViews applet using a typical IDE procedure. The procedures are the same, however, for other IDEs.

This section does not explain the concepts that underlie JavaBeans. For this information, refer to the Web site: `http://java.sun.com/products/javabeans`. Furthermore, it is assumed that you are familiar with the manipulation of JavaBeans within your IDE.

Creating the Manager View

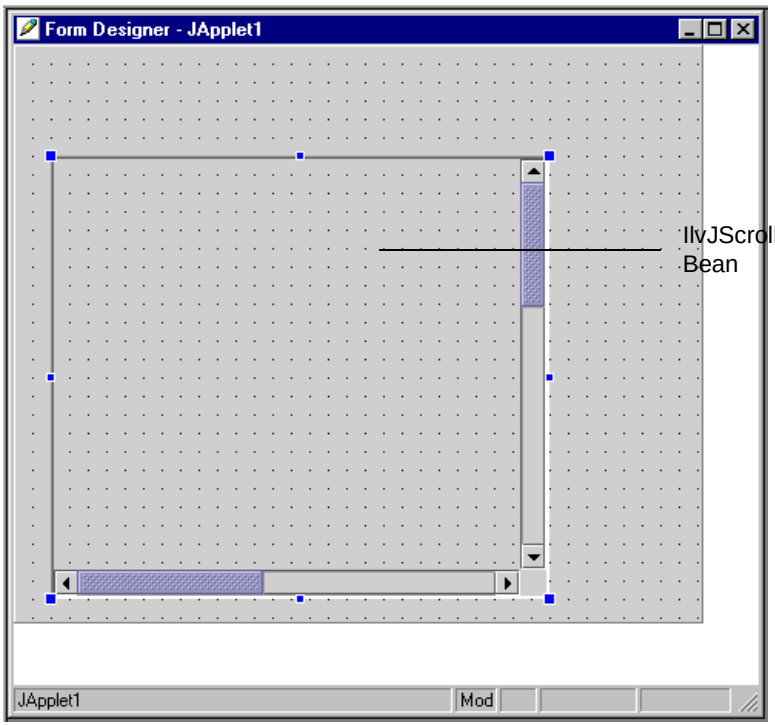
To create the manager view, do the following:

1. Create a new project as a JFC/Swing applet or application.
2. On the Beans toolbar, click the JViews tab to display the JViews Graphics Framework Beans.
3. On the JViews Graphics Framework Beans toolbar, click the `IlvJScrollManagerView` Bean icon and drag it to the Form Designer.

> ***Note:*** *You will notice two scroll manager view icons on the toolbar. Place the pointer over the icon to read the name and choose the one with "J" in its prefix,* `IlvJScrollManagerView`*.*

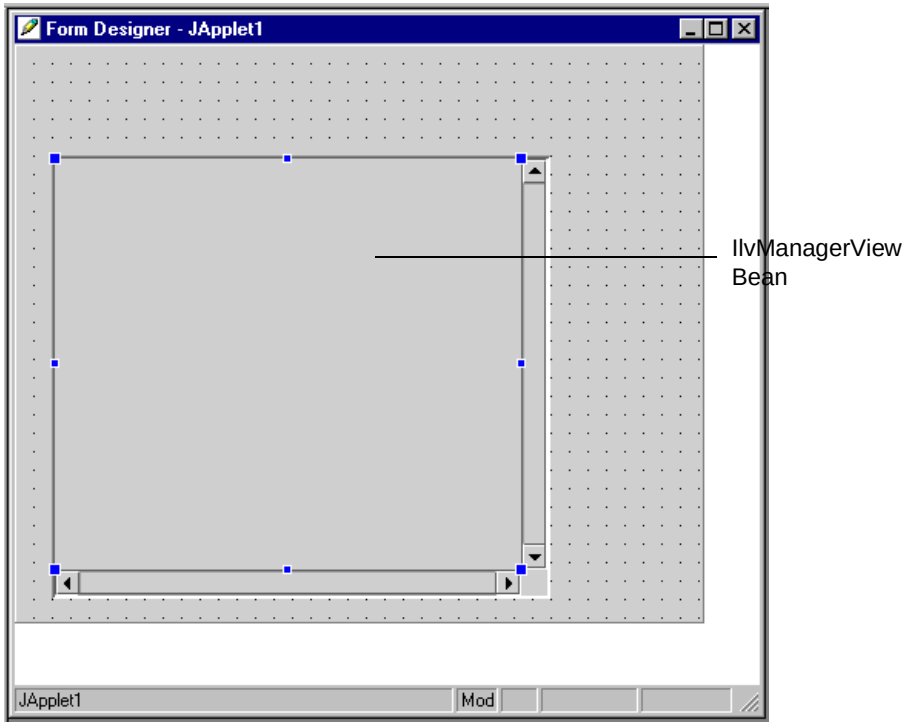
> ***Note:*** *The Graphics Framework Swing Beans have the letter "**J**" in the prefix of the Bean name. You could also create the same type of application using only AWT controls. To do so, you would simply use the* `IlvScrollManagerView` *Bean that is an AWT control instead of the* `IlvJScrollManagerView` *Bean. However, there is no AWT equivalent of the* `IlvJGraphLayoutProgressBar` *Bean.*

4. Drag the handles of the `IlvJScrollManagerView` Bean until it looks approximately like this:



5. On the JViews Beans toolbar, click the `IlvManagerView` icon and drag it inside the `IlvJScrollManagerView` Bean.

   The result is fairly similar to what you obtained in the previous step, except that you can now select the manager view.



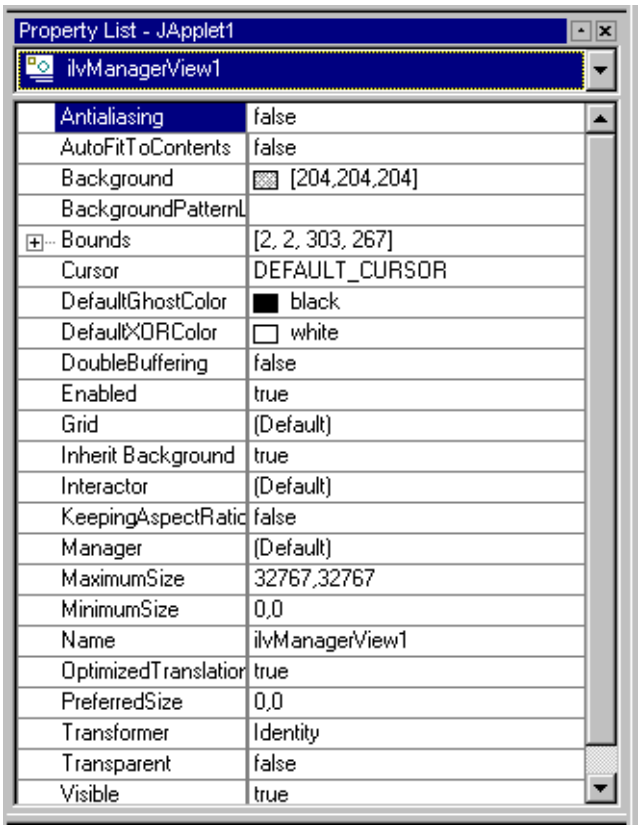
Note: *If you were to compile and run the project at this point, you would see that the* `IlvJScrollManagerView` *allows you to scroll through the contents of the* `IlvManagerView` *Bean. (At the moment, the manager view is empty, so there is nothing to scroll.)*

Setting the Properties of the Manager View

You are now going to change two properties of the `IlvManagerView` Bean. You want to ensure that the double-buffering mechanism will be used and that the zoom level of the manager views will always remain the same along the x-axis and y-axis.

1. Make sure that the `IlvManagerView` Bean is selected in the Form Designer.

   When the `IlvManagerView` Bean is selected in the Form Designer, its properties are displayed in the Property List.



2. Click the value field of the `DoubleBuffering` property. Change the value from `false` to `true`. This will ensure that the double-buffering mechanism will be used.

3. Click the value field of the `KeepingAspectRatio` property. Change the value from `false` to `true`. This will ensure that the zoom level remains the same along the x-axis and y-axis.

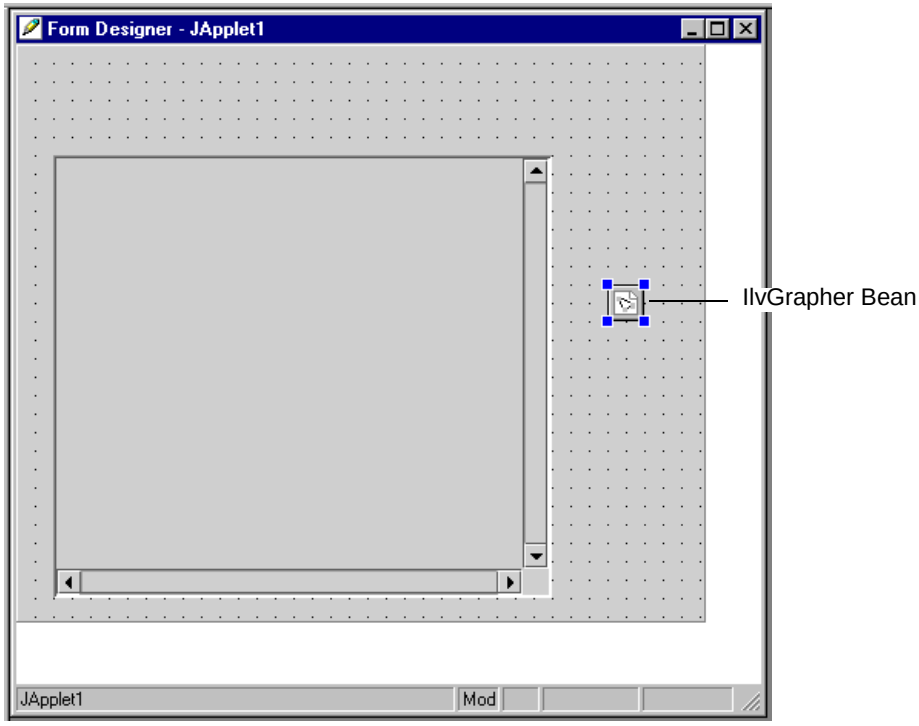
Creating a Grapher and Associating a Manager View

You are now going to create an `IlvGrapher` Bean and associate a manager view with it. The `IlvGrapher` Bean provides the data structure that contains the grapher you want to display.

To create the `IlvGrapher` Bean and associate a manager view with the grapher, do the following:

1. On the JViews Graphics Framework Beans toolbar, click the `IlvGrapher` icon and drag it into the Form Designer.

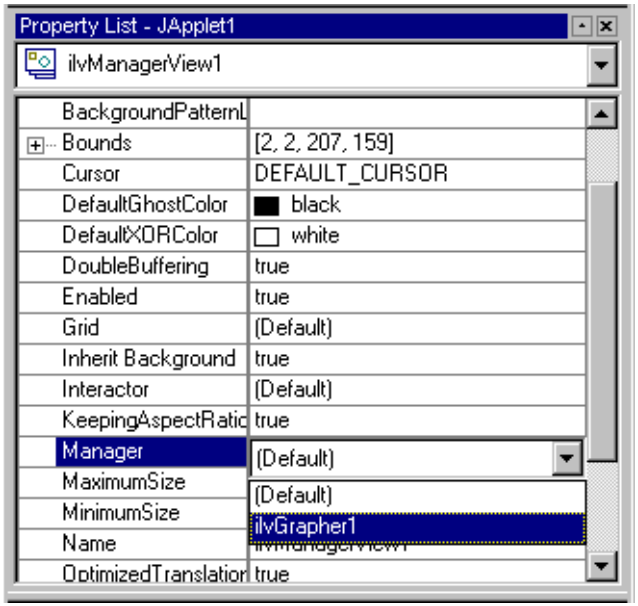
   The `IlvGrapher` class is not a graphical Bean, so it is not managed the same way by the various IDEs. The following figure shows the grapher as a small object inside the Form Designer.



2. To associate the manager view with the grapher, select the `IlvManagerView` Bean so that its properties appear in the Property List.

   You now need to set the `Manager` property of the `IlvManagerView` Bean to the new `IlvGrapher` Bean. (Keep in mind that the `IlvGrapher` class is a subclass of `IlvManager`.)

3. In the Property List window, click the value column of the `Manager` property. Change `Default` to `ilvGrapher1`.



Once this operation is done, the `IlvManagerView` can display the contents of the `IlvGrapher` Bean. You could create several `IlvManagerView` Beans and associate them with the same `IlvGrapher` Bean. This allows you to have several views of the same graph.

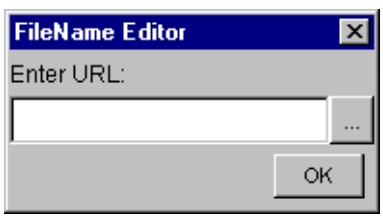
Loading an .ivl File into the Grapher

You are now going to load an `.ivl` file into the `IlvGrapher` Bean so that the contents of the `.ivl` file are displayed in the manager view.

To load an `.ivl` file, do the following:

1. In the Form Designer, select the `IlvGrapher` Bean so that its properties appear in the Property List.
2. Click the value field of the `FileName` property and then click the ··· button.

   The FileName Editor dialog box appears.



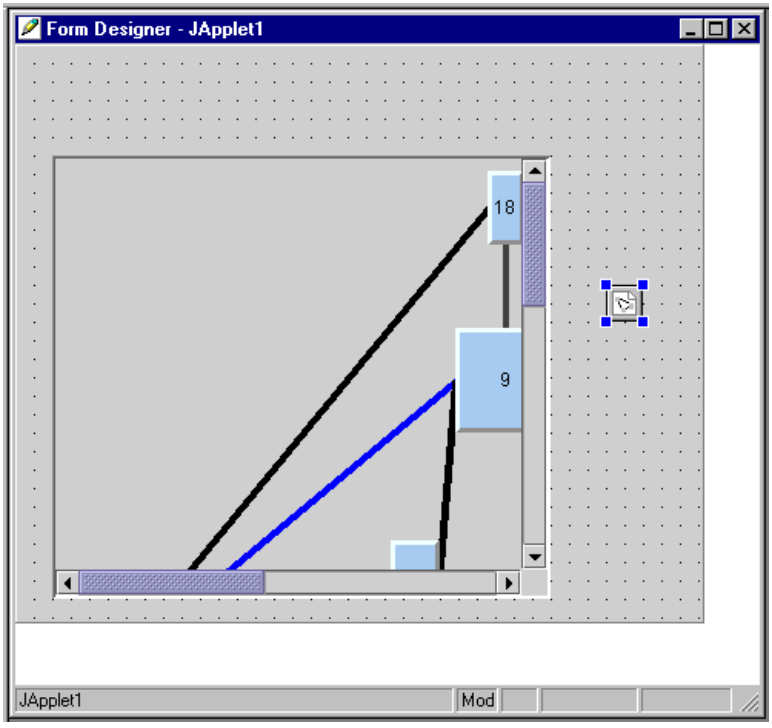
3. To specify the `.ilv` file, click the ... button in the FileName Editor.

   The Choose URL dialog box appears.

4. Go to the `<installdir>/data/graphlayout/link` directory and select the `sample1.ivl` file. Click Open.

5. Click OK in the FileName Editor dialog box.

   The file is automatically displayed in the `IlvManagerView` Bean. As you can see, only a portion of the graph is visible in the manager view.

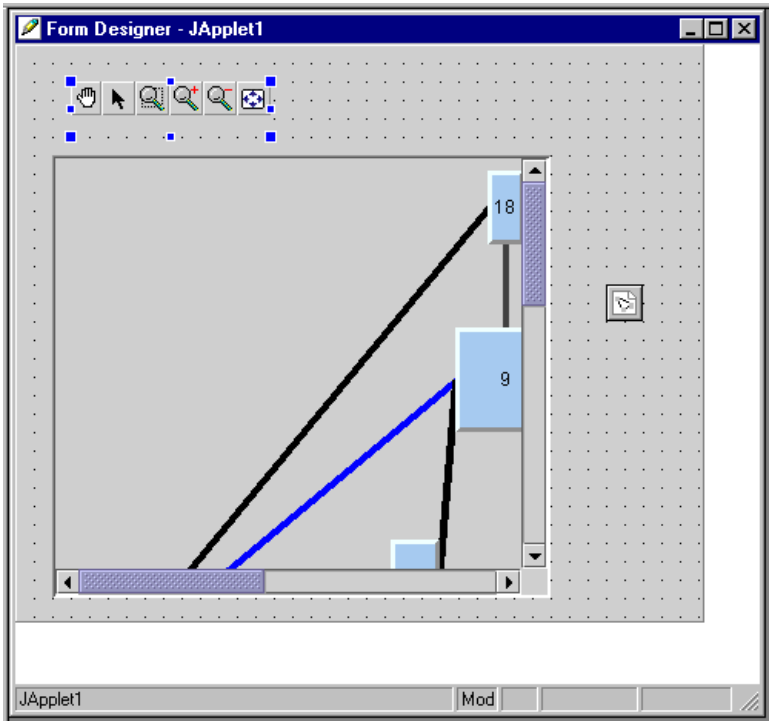


Adding a Control Toolbar Bean

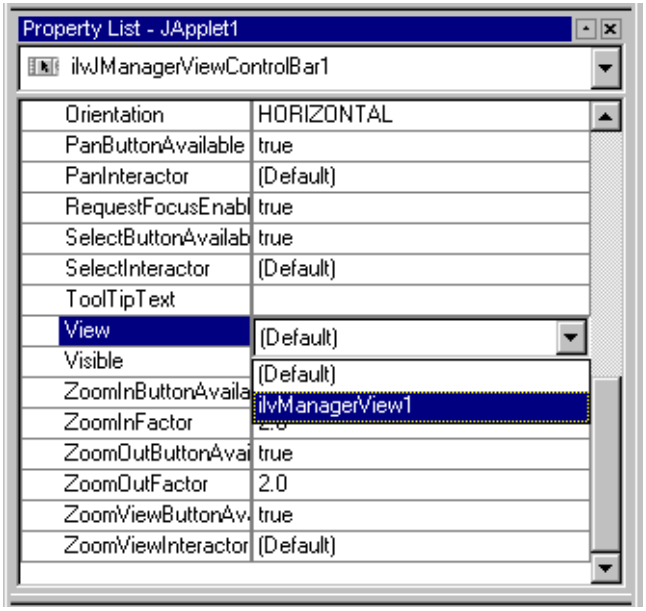
You are now going to add a toolbar and associate the toolbar with the manager view. The toolbar allows the user to control the zoom level of the view and to pan the view.

Do the following:

1. On the JViews Graphics Framework Beans toolbar, click the `IlvJManagerViewControlBar` icon and drag it to the Form Designer.



2. Make sure that the `IlvJManagerViewControlBar` Bean is selected in the Form Designer so that its properties appear in the Property List.
3. To associate the toolbar with the manager view, click the value field of the `View` property and select `ilvManagerView1`.



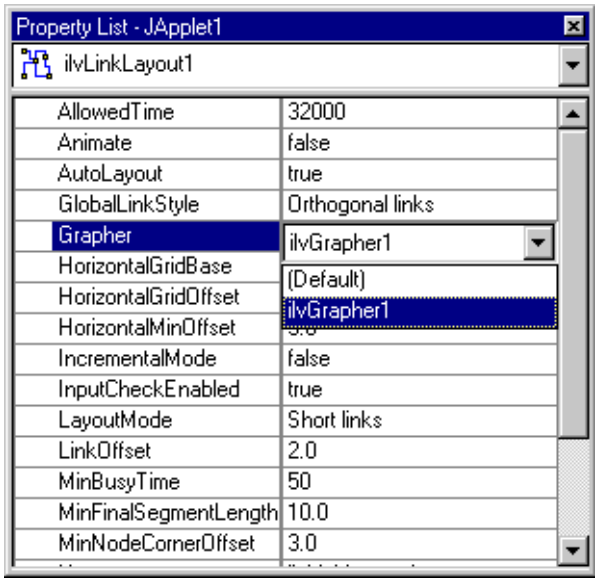
> ***Note:*** *In this example, you added interaction to the view using the control toolbar. You could also set an interactor Bean, such as the* `IlvSelectInteractor`*, directly on the manager view by using the* `interactor` *property of the* `IlvManagerView` *Bean.*

Adding a Graph Layout Bean

You are now going to add the capability to apply a link layout to the graph displayed in your applet.

1. On the Beans toolbar, click the JViews Graph Layout tab to display the JViews Graph Layout Beans.
2. On the JViews Graph Layout Beans toolbar, click the `IlvLinkLayout` Bean icon and drag it to the Form Designer. Place it under the `IlvGrapher` Bean.

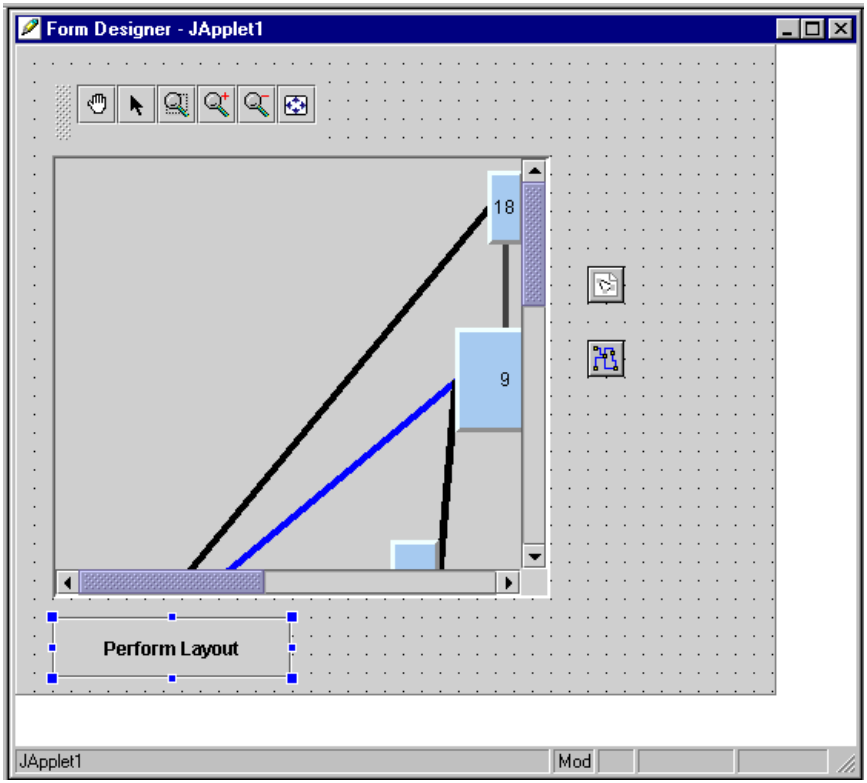
   You are now going to associate the `IlvLinkLayout` Bean with the `IlvGrapher` object.
3. Make sure that the `IlvLinkLayout` Bean is selected in the Form Designer. Its properties should be displayed in the Property List.
4. In the Property List window, click the value field of the `Grapher` property. Change `Default` to `ilvGrapher1`. This indicates to the layout Bean which grapher is to be laid out.



Adding a Swing Bean

You are now going to add a Swing Bean to the applet. This Bean will allow you to launch the layout process on the graph.

1. On the Beans toolbar, click the Swing tab to display the Swing Beans.
2. On the Swing Beans toolbar, click the JButton Bean icon and drag it to the Form Designer. Place this button below the `IlvJScrollManagerView` bean.
3. Make sure that the `JButton` Bean is selected in the Form Designer. Its properties should be displayed in the Property List.
4. In the Property List window, click the value field of the `Text` property. Change `jbutton` to `Perform Layout`.

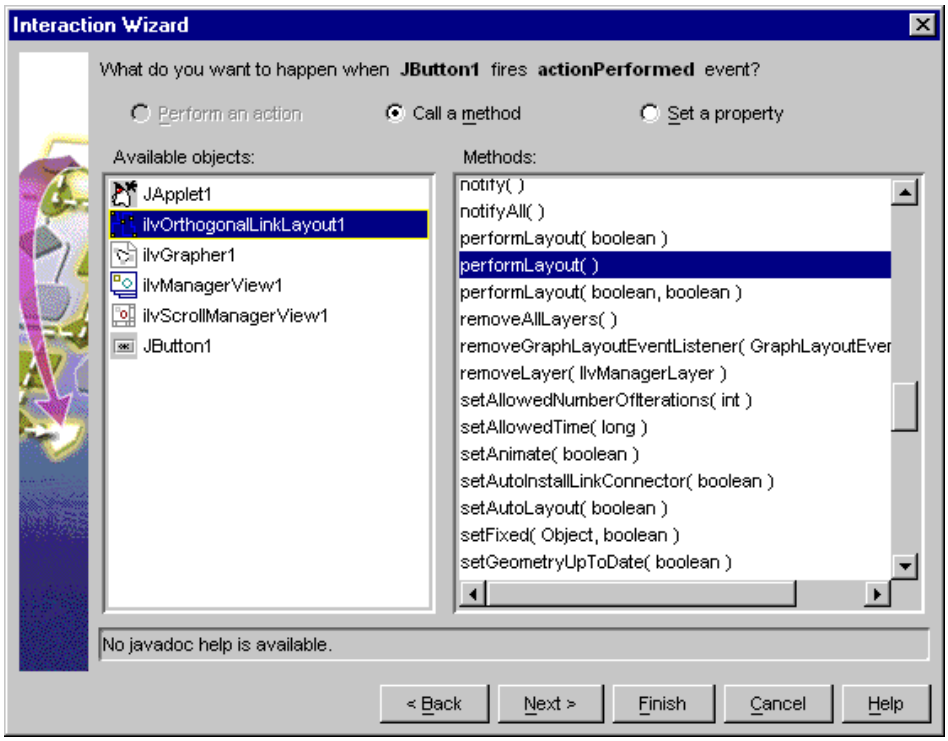


Adding User Interaction

You are now going to define the action that will be performed when the user clicks the Perform Layout button.

1. With the right mouse button, click the `JButton` Bean in the Form Designer.
2. From the pop-up menu that appears, select Add interaction.

   The Interaction Wizard appears.
3. Click Next on the first page of the wizard.
4. In the Events area, make sure that `actionPerformed` is selected. Click Next.
5. On the next page of the Interaction Wizard, click “Call a method”.
6. In the Available objects area, select `IlvLinkLayout1`.
7. In the Methods area, select `performLayout()`.



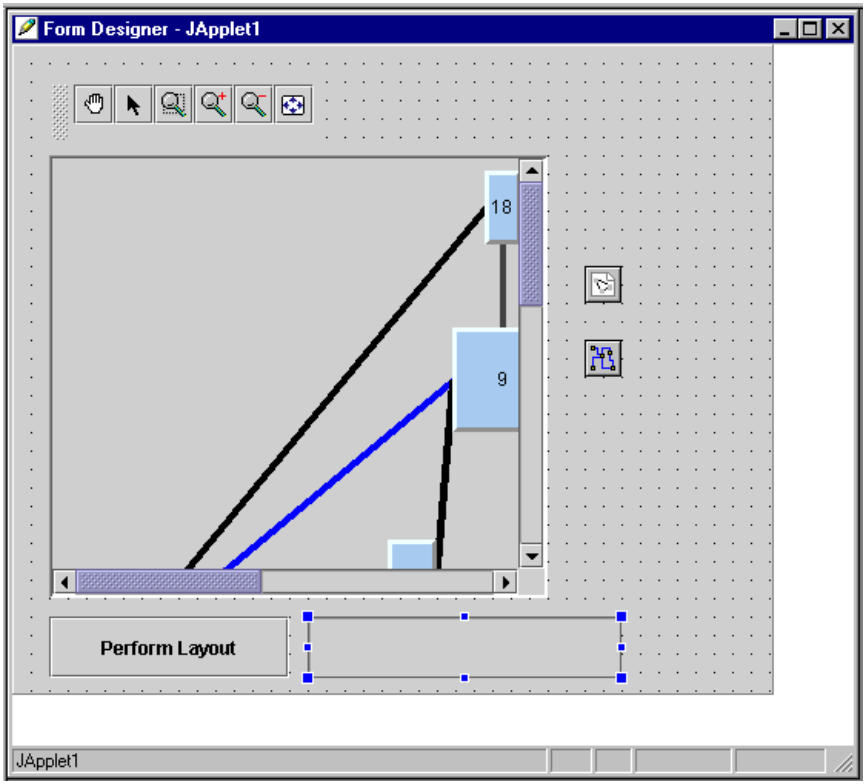
8. Click Finish to apply the interaction. The Interaction Wizard closes.

   Now when the user clicks the Perform Layout button, the layout is performed and the links of the graph are reshaped to orthogonal. (The links appear as alternating horizontal and vertical segments.)

Adding a Progress Bar to the Applet

In some cases, a layout algorithm may take a long time to perform a layout and it may be useful to keep the user informed of the activity of the layout. You are now going to add a progress bar to your applet. (Of course, this step is not mandatory.)

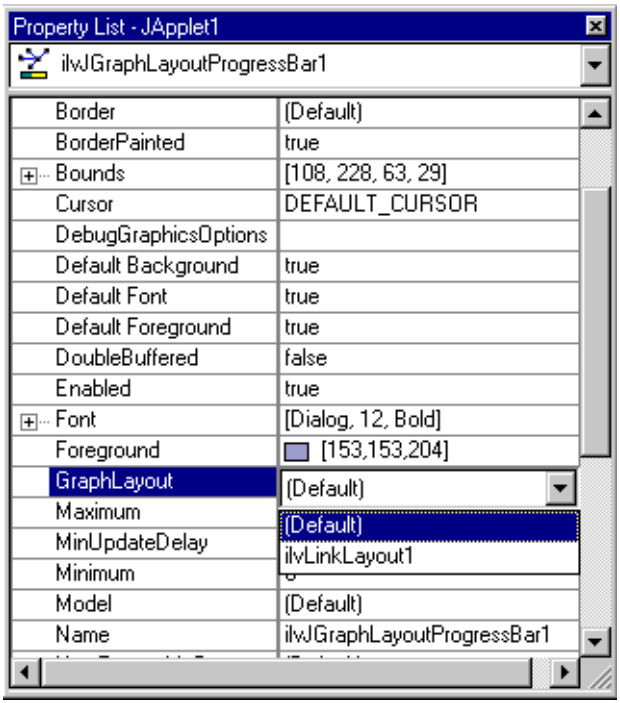
1. On the Beans toolbar, click the JViews Graph Layout tab to display the JViews Graph Layout Beans.
2. On the JViews Graph Layout Beans toolbar, click the `IlvJGraphLayoutProgressBar` Bean icon and drag it to the Form Designer. Place in under the `IlvJScrollManagerView` Bean next to the Perform Layout button.



You are now going to associate the `IlvLinkLayout` Bean with the `IlvJGraphLayoutProgressBar` object.

3. Make sure that the `IlvJGraphLayoutProgressBar` Bean is selected in the Form Designer. Its properties should be displayed in the Property List.

4. In the Property List window, click the value field of the `GraphLayout` property. Change `Default` to `ilvLinkLayout1`.

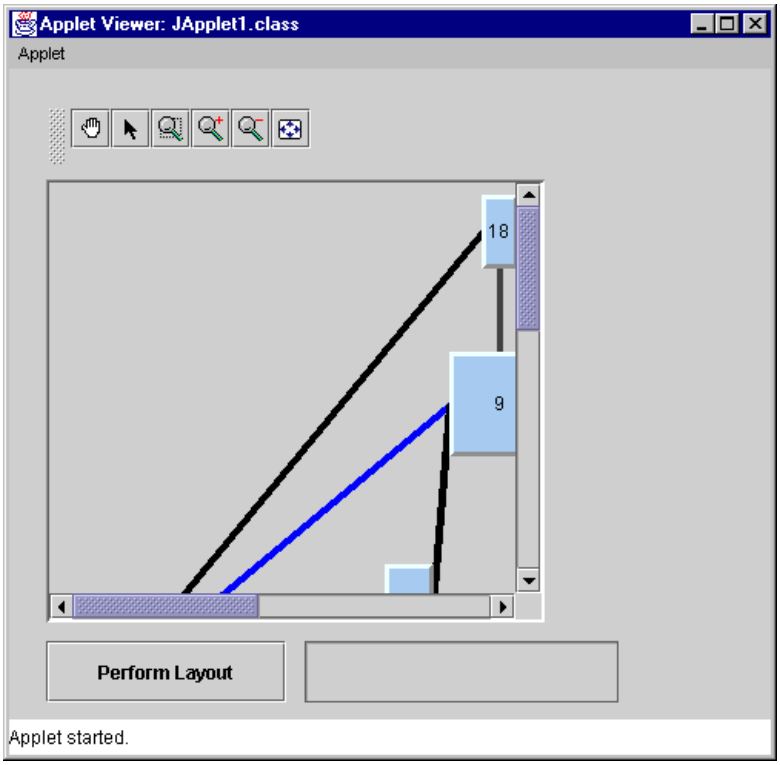


Testing the Result

Now that the applet has been created, you can test the result.

1. Execute the applet. From the Project menu, choose Execute.

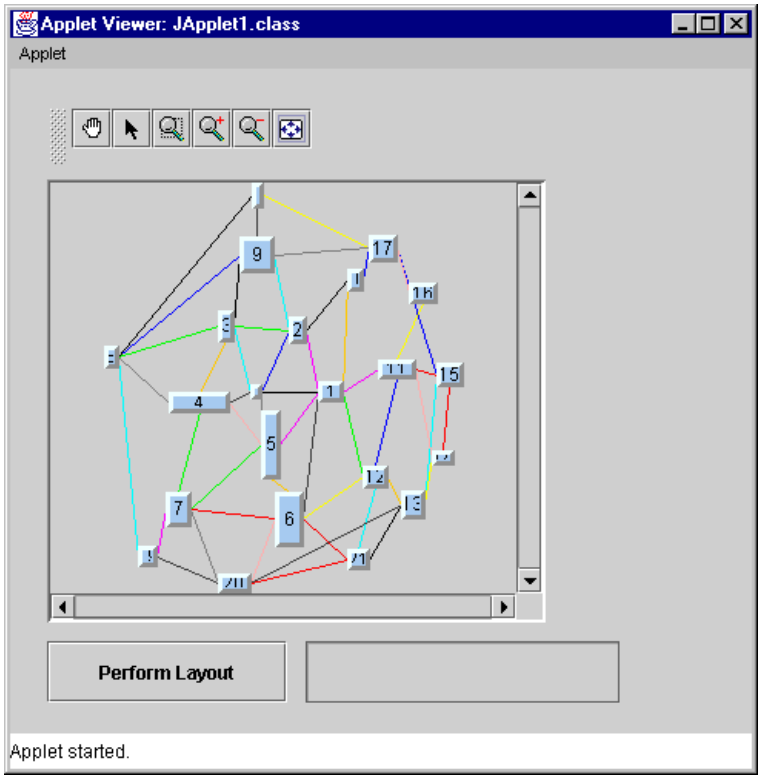
   Initially, the resulting application should appear as shown in the following image (the graph is not yet laid out).



2. You can use the icons in the toolbar to manipulate the graph displayed in the manager view. The toolbar contains the following icons:

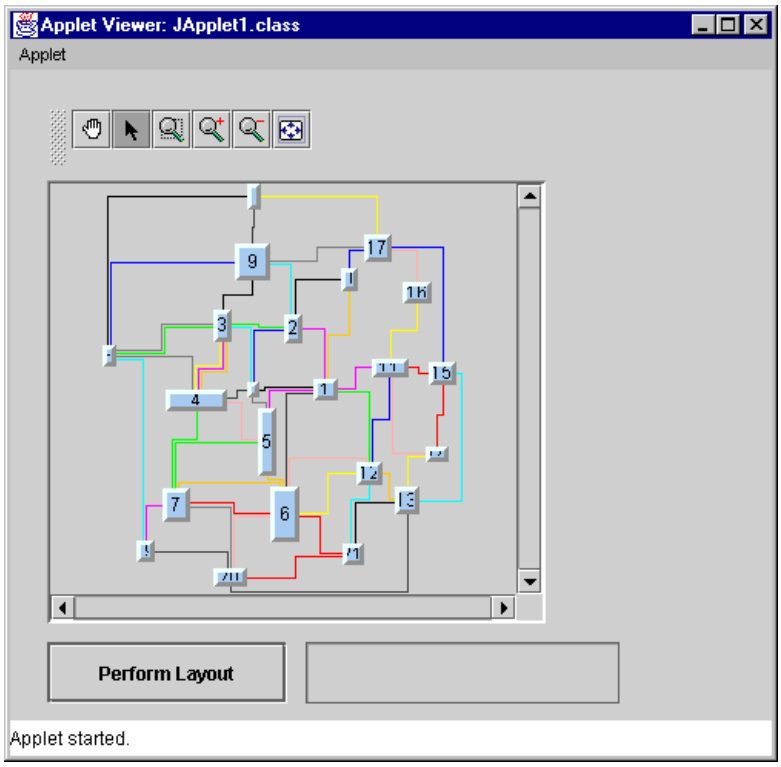
   - The Pan icon to pan the content of a view.
   - The Select arrow icon to select and edit objects in the view.
   - The Interactive zoom icon to drag a rectangle over an area that you want to zoom.
   - The Zoom-in and the zoom-out icons to view the graph at different zoom levels.
   - The Fit-to-content icon to size the graph so that it fits entirely in the manager view.

   When you click the Fit-to-content icon, the applet window appears as shown in the following image.



3. To lay out the graph, click the Perform Layout button.

   As the layout is being performed, you should see the progress indicated in the progress bar at the bottom of the window. When the layout is completed, the applet window should appear similar to the window in the following image.

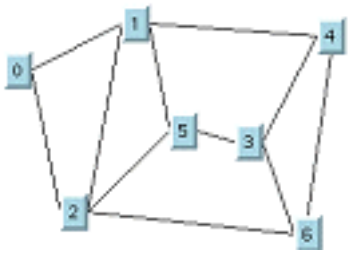


This completes the Graph Layout Beans example. For information on how to save your project along with the type of files that are generated when saving, refer to the documentation for your IDE.

Glossary

2-connected graph

Also known as a biconnected graph: A connected graph that will remain a connected graph if a discretionary node is removed. (In other words, a graph in which at least two distinct paths exist between every two nodes.)



acyclic graph

A graph that has no cycles.

automatic layout

A layout process where the layout algorithm does everything without any user intervention.

biconnected graph

See 2-connected graph.

bus topology

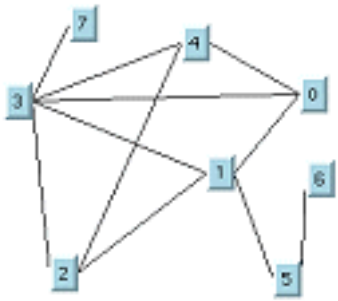
A type of network topology where a set of nodes is connected to a bus object.

connected component

A connected graph or subgraph. A connected component of graph G is a maximal connected subgraph of G.

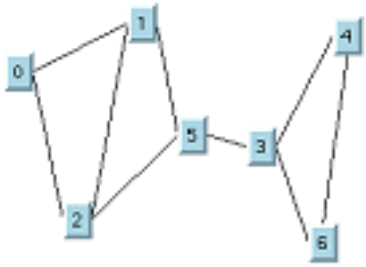
connected graph

A graph where there is a path connecting each pair of nodes. See also disconnected graph.

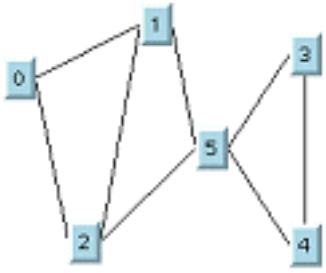


connection Another name for an edge of a graph. See also edge.

cut-edge An edge is a cut-edge if the graph is no longer connected when the edge is removed from the graph. In the following illustration, the link between nodes 3 and 5 is a cut-edge. Note that its end points are cut-nodes.



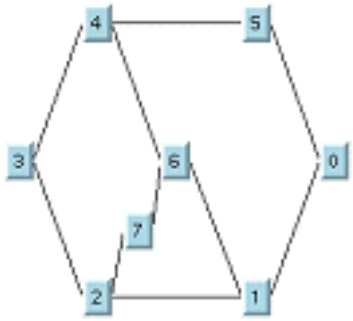
cut-node A node is a cut-node if the graph is no longer connected when the node is removed from the graph. In the following illustration, node 5 is a cut-node.



cycle A path of a graph that begins and ends on the same node. Also called a loop.

cyclic graph A graph that contains cycles.

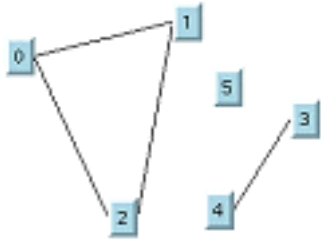
cyclomatic number A number equal to m - n + 1, where m = number of edges and n = number of nodes. In the following illustration, the number of edges = 10, the number of nodes = 8. Therefore, the cyclomatic number is 10 - 8 + 1 = 3.



degree The number of neighbors of a node.

directed graph A graph in which every edge is associated with an ordered pair of nodes. Also called an oriented graph.

disconnected graph A graph containing at least two nodes are not linked by a path. In the following illustration, node 5 is not linked by a path to any other node. Nodes 3 and 4 are not linked by a path to any nodes other than nodes 3 and 4.



edge A line connecting two vertices in a graph. An edge is also called a link or a connection. In this documentation, the term link is primarily used for the term edge.

edge crossing See link crossing.

fixed nodes If the user specifies a node as fixed, layout algorithms are not allowed to move this node.

fixed links — If the user specifies a link as fixed, layout algorithms are not allowed to reshape this link.

force-directed layout — A class of layout algorithms for creating straight-line drawings of undirected graphs based on an iterative computation of the position of the nodes according to a set of attractive and repulsive "forces." These forces are computed in such a way that they tend to produce a layout with only a few link crossings or no link crossings. The Spring Embedders are an example of force-directed layout algorithms.

graph — A finite set of nodes (also called vertices) connected by a finite set of links (also called edges or connections).

grapher — In ILOG JViews, an instance of the class `IlvGrapher`. More generally, a Java object that is used to manage a collection of nodes and links.

graph layout — The process that applies a layout algorithm to a graph. Also the graph drawing that results from the layout process.

graph model — In the ILOG JViews Graph Layout framework, a Java class that defines a suitable, generic API for graphs or an instance of the class (`IlvGraphModel`).

grid drawing — A drawing where nodes and link bends have discrete (integer) coordinates.

incremental layout — A layout process where the result of a previous layout is used as the starting point for applying the layout algorithm a second time to a modified graph in order to minimize the changes.

incident — A link-node connection. A link is incident to a node if the node is at one end of the link. A node is incident to a link if that link is incident to the node.

layer — In ILOG JViews, the storage area of a manager in which graphic objects are placed. A manager uses multiple layers, referenced by index numbers. Objects in a higher-numbered layer are displayed in front of objects in a lower-numbered layer.

layout algorithm — The process that computes new coordinates for nodes and/or new shapes for links in order to obtain a suitable representation of a graph.

layout region — In the ILOG JViews Graph Layout module, the rectangle where the graph drawing will be placed when the graph is laid out.

link Another name for an edge in a graph. In this documentation, the term link is primarily used for edge. See also *edge*.

link connector In ILOG JViews, the system that allows you to control the connections of links to nodes in a grapher.

link crossing Link crossings occur when links intersect at places other than an incident node. Also called edge crossings. Often, layout algorithms are used to minimize the number of link crossings.

listener In ILOG JViews, an interface allowing you to connect events that occur in the manager view with actions to be performed.

loop See cycle.

manager In ILOG JViews, a holding place for grouping graphic objects and for coordinating their behavior and display in multiple layers and multiple views.

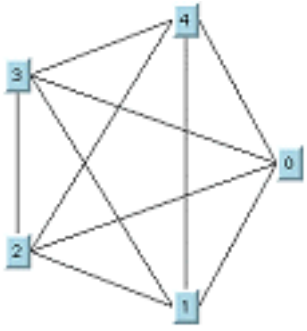
manager view See view.

multiple links More than one link between the same origin and destination nodes.

neighbor nodes The nodes that are connected to a given node by a link.

node Another name for a vertex of a graph. In this documentation, the term node is primarily used for vertex.

nonplanar graph A graph that cannot be drawn without any links crossing other links.

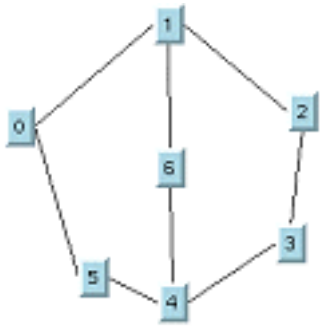


NP-complete A class of computational problems for which no efficient solution algorithm has been found. Many significant computer science problems, including many graph layout problems, belong to this class.

orthogonal drawing A drawing where each link is drawn as a polygonal chain of alternating horizontal and vertical segments.

path A sequence of consecutive nodes leading from one node to another using the links of the graph. It is the route along the links through the nodes in a graph. The length of a path is the number of links traversed.

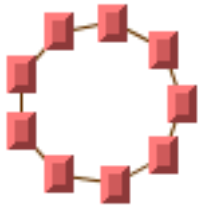
planar graph A graph that can be drawn with no links crossing other links.



polyline drawing A drawing where each link is drawn as a polygonal chain.

radial drawing A layout style where the nodes are placed radially around a root node.

ring A type of network topology where the nodes are arranged in a circular configuration.



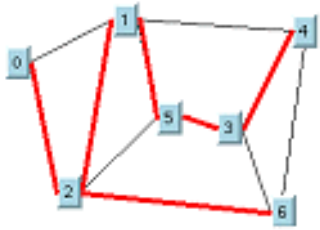
seed value The value that is used for the initialization of a random number generator. Some layout algorithms use random numbers during the layout computation.

self-link A link whose origin and destination nodes are the same node.

semiautomatic layout A layout process where the user makes manual improvements to the result of the automatic layout process.

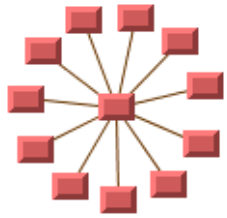
spanning tree

A minimal subgraph, defined as follows: A spanning tree S of a graph G is a subgraph of G containing all the nodes of the graph and whose links are a subset of the links of the graph. The number of links of G that are not present in S must be the minimum number for which there are no cycles in S. The spanning tree is shown by the red links in the following illustration.



star

A type of network topology where the nodes are arranged on a circle with each node being connected to a center node.



static layout

A layout process where the layout algorithm is completely redone each time the graph is changed.

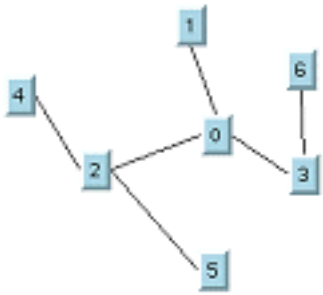
straight-line drawing

A drawing where each link is drawn as a straight line segment.

subgraph

G' is a subgraph of G if its node and link sets are included in the node and link sets of G.

tree A connected acyclic graph (that is, a graph that does not contain any cycles).



topology The structure of a graph. Two drawn graphs have the same topology if you can obtain one drawing from the other by moving the nodes and reshaping the links.

undirected graph A graph in which every link is associated with an unordered pair of nodes.

vertex A “dot” in a graph. A graph consists of a finite set of vertices connected by a finite set of edges (also called links or connections). A vertex is also called a node. In this documentation, the term node is primarily used for the term vertex.

view In ILOG JViews, the AWT component where graphic objects are displayed. To display graphic objects contained in the different layers of a grapher, you create at least one view, and often multiple views. The grapher lets you connect as many views as you require to display graphic objects.

Index

A

absolute level position range **167**
absolute level position tendency **167**
`accept` method
  `IlvLayoutGraphicFilter` class **322**
accessing
  all sublayouts **275**
`addClusterId` method
  `IlvCircularLayout` class **246**
`addGraphLayoutEventListener` method
  `IlvGraphLayout` class **295**
`addGraphModelListener` method
  `IlvGraphModel` class **301**
adding
  layout constraints **171**
`addLabelingModelListener` method
  `IlvLabelingModel` class **387**
`addLayer` method
  `IlvDefaultLabelingModel` class **379**
  `IlvLayoutGraphicFilter` class **322**
`addLineObstacle` method
  `IlvLongLinkLayout` class **214**
`addNode` method
  `IlvGrapher` class **341**
`addObject` method
  `IlvManager` class **341**
`addRectObstacle` method
  `IlvLongLinkLayout` class **214**
`adjustmentEnd` method
  `IlvGraphModel` class **301**
alignment options
  Grid Layout **259**
  Tree Layout (free mode) **106**
allowed time parameter
  Bus Layout **224**
  Grid Layout **257**
  Hierarchical Layout **140**
  Link Layout **191**
  Multiple Layout **285**
  Recursive Layout **271**
  Spring Embedder Layout **75**
  Topological Mesh Layout **59**
  Tree Layout **99**
  Uniform Length Edges Layout **85**
angle layout criteria **15**
animation parameter
  Link Layout **191**
  Spring Embedder Layout **75**
  Topological Mesh Layout **59**
  Uniform Length Edges Layout **85**
Annealing Label Layout **354**
  allowed time parameter **358**
  automatic update **372**
  code sample **357**
  description **356**
  expert parameters **372**
  features **356**
  generic parameters **358**
  global parameters **369**

implementing your own label descriptors **373**
label descriptors **359**
subclasses **360**
label movement policy **371**
label offset parameter **369**
limitations **356**
obstacle offset parameter **370**
percentage completion calculation parameter **358**
point label descriptor **360**
polyline label descriptor **364**
random generator seed value parameter **358**
sample drawings **354**
save parameters to named properties **359**
stop immediately parameter **359**
use default parameters **359**
area layout criteria **14**
area minimization parameter, Circular Layout **247**
aspect ratio parameter
Tree Layout (radial mode) **122**
Tree Layout (tip-over mode) **126**
`attach` method
`IlvGraphLayout` class **18**
`IlvLabelLayout` class **348**
attaching a grapher **18**
automatic label placement **339**, **340**
automatic layout
description of **15**, **49**
automatic layout parameter
Link Layout **191**
automatic update
labels **372**

B

Beans
associating objects **397**
classes available as Beans **392**
creating a graph layout Bean **402**
creating a grapher **397**
creating a toolbar **400**
creating the manager view **394**
editing properties **396**
example applet **393**
executing the applet **407**
`IlvJScrollManagerView` class **394**
`IlvManagerView` class **395**
loading an `.ivl` file **399**
setting properties of the manager view **396**
testing the applet **407**
bends layout criteria **15**
`boundingBox` method **380**
`IlvGraphic` class **324**
`IlvGraphModel` class **300**
`IlvLabelingModel` class **386**, **390**
Bus Layout
applicable graph types **222**
application domains **222**
bus node parameter **228**
code sample **223**
description **223**
dimensional parameters **234**
features **223**
flow direction parameter **229**
generic parameters **224**
global alignment parameters **232**
horizontal offset parameter **234**
incremental mode parameter **234**
individual node alignment parameter **233**
link clipping parameter **225**
link style parameter **228**
margin on bus parameter **235**
margin parameter **235**
maximum nodes per level parameter **230**
order parameter **226**
ordering options table **227**
sample drawing **222**
specific parameters **226**
vertical offset parameter **235**
vertical offset to previous level parameter **235**
width adjusting **231**
bus node parameter, Bus Layout **228**
bypass distance parameter, Link Layout (short link mode) **210**

C

calculated level index parameter, Hierarchical Layout **183**
calculated position index parameter, Hierarchical Layout **184**
calculating link shapes, Tree Layout **98**
calculating node positions, Tree Layout **98**

calculating the spanning tree, Tree Layout **97**
Circular Layout
  applicable graph types **239**
  application domains **239**
  area minimization parameter **247**
  cluster contents parameter **251**
  cluster membership parameters **245**
  cluster position parameter **251**
  cluster size parameter **251**
  clustering mode parameter **245**
  code sample **242**
  description **240**
  dimensional parameters **250**
  disconnected graph offset parameter **251**
  features **240**
  generic parameters **243**
  level offset parameter **250**
  limitations **240**
  link clipping parameter **244**, **252**
  link connection box parameter **244**, **253**
  link style parameter **252**
  offset parameter **250**
  order of nodes parameters **245**
  ring topology **241**
  root clusters parameter **247**
  sample drawings **238**
  specific parameters **245**
  star center parameter **247**
  star topology **241**
cluster contents parameter, Circular Layout **251**
cluster membership parameters, Circular Layout **245**
cluster position parameter, Circular Layout **251**
cluster size parameter, Circular Layout **251**
clustering modes
  Circular Layout **245**
compass directions, Tree Layout **103**
connected components parameter
  Bus Layout **224**
  Circular Layout **244**
  Hierarchical Layout **141**
  Multiple Layout **285**
  Spring Embedder Layout **75**
  Topological Mesh Layout **59**
  Tree Layout **100**
  Uniform Length Edges Layout **85**
connector style parameter
  Hierarchical Layout **147**
  Tree Layout **111**
coordinates mode
  Label Layout **380**
  specifying **327**
core library of ILOG JViews **16**
`createGraphLayout` method
  `IlvDefaultLayoutProvider` class **280**, **312**
`createGraphModel` method
  `IlvGraphModel` class **300**
`createLayoutReport` method
  `IlvGraphLayout` class **292**
creating a layout report **292**

D

default labeling model
  subclassing **388**
defining a new type of layout **329**
`detach` method
  `IlvGraphLayout` class **20**
  `IlvLabelLayout` class **19**, **348**
detaching a grapher **20**
`detachLayouts` method
  `IlvDefaultLayoutProvider` class **276**, **312**
dimensional parameters
  Bus Layout **234**
  Grid Layout **261**
dimensional parameters, Circular Layout **250**
disconnected graph
  laying out connected components **304**
disconnected graph offset parameter, Circular Layout **251**

E

east-west neighbors, Tree Layout **129**
end points mode parameter
  Hierarchical Layout **148**
  Link Layout **197**
events
  label layout **352**
extremity constraints, Hierarchical Layout **177**

F

fallback mechanism, Link Layout (long link mode) **215**
features
  generic parameter descriptions **8**
  layout algorithms **4**
fields of application for graph layouts **10**
filtering features
  by graphic objects **323**
  by layers **322**
  for `IlvGrapher` **321**
`fireGraphModelEvent` method
  `IlvGraphModel` class **301**
`fireLabelingModelEvent` method
  `IlvLabelingModel` class **387**
fixed links parameter
  Bus Layout **225**
  Circular Layout **244**
  Hierarchical Layout **141**
  Link Layout **191**
  Random Layout **219**
  Spring Embedder Layout **76**
  Topological Mesh Layout **60**
  Tree Layout **100**
  Uniform Length Edges Layout **86**
fixed nodes parameter
  Bus Layout **225**
  Circular Layout **244**
  Grid Layout **258**
  Hierarchical Layout **141**
  Random Layout **219**
  Spring Embedder Layout **76**
  Topological Mesh Layout **60**
  Tree Layout **101**
  Uniform Length Edges Layout **86**
  using to refine a TML graph layout **66**
flow direction parameter
  Bus Layout **229**
  Hierarchical Layout **142**
  Tree Layout (free mode) **104**
force fit to layout region, Uniform Length Edges Layout **88**
fork link shapes
  Hierarchical Layout **156**
free layout mode (Tree Layout)
  alignment parameter **106**
  description **104**
  flow direction **104**
  global alignment **106**
  global link style parameter **110**
  individual link style **110**
  individual node alignment **107**
  link style **109**
  orthogonal fork percentage **117**
  respect node sizes **117**
  spacing parameters **115**
  spacing parameters for experts **117**
  tip-over alignment **108**

getA

`getAlignment` method
  `IlvTreeLayout` class **108**
`getAllowedNodesPlacementTime` method
  `IlvTopologicalMeshLayout` class **62**
`getAllowedNumberOfIterations` method
  `IlvUniformLengthEdgesLayout` class **87**
`getAllowedNumberOfNodesPlacementIterations` method
  `IlvTopologicalMeshLayout` class **62**
`getAllowedNumberOfOptimizationIterations` method
  `IlvTopologicalMeshLayout` class **62**
`getAllowedOptimizationTime` method
  `IlvTopologicalMeshLayout` class **62**
`getAllowedTime` method
  `IlvGraphLayout` class **22**
`getAspectRatio` method
  `IlvTreeLayout` class **123**

getB

`getBottomMargin` method
  `IlvGridLayout` class **263**
`getBox` method
  `IlvLinkConnectionBoxInterface` class **29**, **208**, **211**
  `IlvNodeBoxInterface` class **207**
`getBranchOffset` method
  `IlvTreeLayout` class **117**
`getBus` method

IlvBusLayout class **228**

getC

getCalcBackwardTreeLinks method
IlvTreeLayout class **131**
getCalcFallbackLinks method
IlvLongLinkLayout class **216**
getCalcForwardTreeLinks method
IlvTreeLayout class **131**
getCalcNodeLevelIndex method
IlvHierarchicalLayout class **183**
getCalcNodePositionIndex method
IlvHierarchicalLayout class **184**
getCalcNonTreeLinks method
IlvTreeLayout class **131**
getCalcRoots method
IlvTreeLayout class **102**
getClusterCenter method
IlvCircularLayout class **251**
getClusterIds method
IlvCircularLayout class **247**, **251**
getClusterIdsCount method
IlvCircularLayout class **247**
getClusterNodes method
IlvCircularLayout class **251**
getClusterRadius method
IlvCircularLayout class **251**
getCode method
IlvGraphLayoutReport class **294**
IlvLabelLayoutReport class **352**
getConnectorStyle method
IlvHierarchicalLayout class **148**, **156**
IlvTreeLayout class **113**
getCoordinatesMode method
IlvGrapherAdapter class **328**

getD

getDestinationPointMode method
IlvHierarchicalLayout class **149**
IlvLinkLayout class **198**
getDisconnectedGraphOffset method
IlvCircularLayout class **251**

getE

getEastNeighbor method
IlvTreeLayout class **130**
getExteriorCycleId method
IlvTopologicalMeshLayout class **64**

getF

getFilter
IlvLayoutGraphicFilter class **322**
getFirstGraphLayout method
IlvMultipleLayout class **287**
getFlowDirection method
IlvBusLayout class **230**
IlvHierarchicalLayout class **143**
IlvTreeLayout class **106**
getFrom method
IlvGraphModel class **300**
getFromPortIndex method
IlvHierarchicalLayout class **156**
getFromPortSide method
IlvHierarchicalLayout class **155**

getG

getGlobalAlignment method
IlvBusLayout class **232**
IlvTreeLayout class **107**
getGlobalDestinationPointMode method
IlvHierarchicalLayout class **149**
IlvLinkLayout class **197**
getGlobalHorizontalAlignment method
IlvGridLayout class **259**
getGlobalLinkStyle method
IlvHierarchicalLayout class **146**
IlvLinkLayout class **195**
IlvTreeLayout class **110**
getGlobalOriginPointMode method
IlvHierarchicalLayout class **149**
IlvLinkLayout class **197**
getGlobalSelfLinkStyle method
IlvShortLinkLayout class **209**, **211**
getGlobalVerticalAlignment method
IlvGridLayout class **259**

getGraphLayout method
IlvGraphLayout class **269**, **311**
getGraphModel method
IlvGraphModel class **300**

getH

getHorizontalAlignment method
IlvGridLayout class **260**
getHorizontalGridBase method
IlvLongLinkLayout class **205**
getHorizontalGridOffset method
IlvGridLayout class **262**
IlvLongLinkLayout class **205**
getHorizontalLinkOffset method
IlvHierarchicalLayout class **160**
getHorizontalMinOffset method
IlvLongLinkLayout class **205**
getHorizontalNodeLinkOffset method
IlvHierarchicalLayout class **160**
getHorizontalNodeOffset method
IlvHierarchicalLayout class **160**
getHorizontalOffset method
IlvBusLayout class **235**

getI

getIndex method
IlvBusLayout class **226**
IlvCircularLayout class **246**
getInterGraphLinks method
IlvGraphModel class **300**
getInterGraphLinksCount method
IlvGraphModel class **300**, **304**

getL

getLabelLayout method
IlvMultipleLayout class **287**
getLabelOverlap method
IlvLabelingModel class **386**, **389**
getLabels method
IlvLabelingModel class **385**, **389**
getLabelsCount method
IlvLabelingModel class **385**
getLayers method
IlvDefaultLabelingModel class **379**
IlvLayoutGraphicFilter class **323**
getLayoutMode method
IlvGridLayout class **258**
IlvLinkLayout class **192**, **245**
IlvTreeLayout class **103**
getLayoutOfConnectedComponents method
IlvGraphLayout class **305**
getLayoutRegion method
IlvGraphLayout class **25**
IlvRandomLabelLayout class **379**
getLayouts method
IlvGraphLayout class **308**
getLayoutTime method
IlvGraphLayoutReport class **294**
IlvLabelLayoutReport class **352**
getLeftMargin method
IlvGridLayout class **263**
getLevelAlignment method
IlvTreeLayout class **120**
getLevelJustification method
IlvHierarchicalLayout class **144**
getLevelOffset method
IlvCircularLayout class **251**
getLinkOffset method
IlvShortLinkLayout class **202**
getLinkPointAt method
IlvGraphModel class **300**, **304**
getLinkPoints method
IlvGraphModel class **300**
getLinkPointsCount method
IlvGraphModel class **300**
getLinkPriority method
IlvHierarchicalLayout class **158**
getLinks method
IlvGraphModel class **299**
getLinksCount method
IlvGraphModel class **299**, **303**
getLinksFrom method
IlvGraphModel class **299**
getLinksFromCount method
IlvGraphModel class **299**, **304**
getLinksTo method
IlvGraphModel class **299**

getLinksToCount method
IlvGraphModel class **300**, **304**
getLinkStyle method
IlvBusLayout class **229**
IlvCircularLayout class **252**
IlvHierarchicalLayout class **146**
IlvLinkLayout class **196**
IlvRandomLayout class **221**
IlvSpringEmbedderLayout class **78**
IlvTopologicalMeshLayout class **25**, **61**
IlvTreeLayout class **111**
IlvUniformLengthEdgesLayout class **87**
getLinkWidth method
IlvGraphModel class **300**
getLongLinkLayout method
IlvLinkLayout class **213**

getM

getMargin method
IlvBusLayout class **235**
getMarginOnBus method
IlvBusLayout class **235**
getMaxChildrenAngle method
IlvTreeLayout class **126**
getMaxNumberOfNodesPerLevel method
IlvBusLayout class **230**
getMaxNumberOfNodesPerRowOrColumn method
IlvGridLayout class **260**
getMinFinalSegmentLength method
IlvLongLinkLayout class **206**
IlvShortLinkLayout class **203**
getMinNodeCornerOffset method
IlvLongLinkLayout class **206**

getN

getNeighbors method
IlvGraphModel class **300**
getNodes method
IlvGraphModel class **299**
getNodesAndLinks method
IlvGraphModel class **299**
getNodesCount method
IlvGraphModel class **299**, **303**
getNodesDegree method
IlvGraphModel class **300**
getNodesPlacementAlgorithm method
IlvTopologicalMeshLayout class **63**
getNumberOfPorts method
IlvHierarchicalLayout class **156**
getNumberOfPossibleExteriorCycles method
IlvTopologicalMeshLayout class **64**

getO

getObstacleOverlap method
IlvLabelingModel class **386**, **390**
getObstacles method
IlvLabelingModel class **385**, **389**
getObstaclesCount method
IlvLabelingModel class **385**
getOffset method
IlvCircularLayout class **250**
getOpposite method
IlvGraphModel class **300**
getOrdering method
IlvBusLayout class **226**
getOriginPointMode method
IlvHierarchicalLayout class **149**
IlvLinkLayout class **197**
getOrthForkPercentage method
IlvTreeLayout class **117**
getOverlapPercentage method
IlvTreeLayout class **118**

getP

getParentChildOffset method
IlvTreeLayout class **117**
getParentModel method
IlvGraphModel class **300**
getPenalty method
IlvTerminationPointFilter class **214**
getPercentageComplete method
IlvLabelLayoutReport class **352**
getPolylineObstacleOverlap method
IlvLabelingModel class **386**
getPolylinePoints method
IlvLabelingModel class **386**

getPolylineWidth method
IlvLabelingModel class **386**
getPosition method
IlvTreeLayout class **103**
getPreferredLayout method
IlvDefaultLayoutProvider class **280**, **312**
getPreferredLinksLength method
IlvUniformLengthEdgesLayout class **87**, **88**
getPreferredPathLocation method
IlvAnnealingLabelDescriptor class **375**
getProperty method
IlvGraphModel class **302**
IlvLabelingModel class **388**

getR

getReferenceTransformer method
IlvLayoutGraphicFilter class **326**
getReferenceView method
IlvLayoutGraphicFilter class **325**
getRightMargin method
IlvGridLayout class **263**
getRootClusterId method
IlvCircularLayout class **247**
getRootModel method
IlvGraphModel class **300**

getS

getSecondGraphLayout method
IlvMultipleLayout class **287**
getSeedValueForRandomGenerator method
IlvGraphLayout class **34**
getShortLinkLayout method
IlvLinkLayout class **208**
getSiblingOffset method
IlvTreeLayout class **117**
getSpecNodeLevelIndex method
IlvHierarchicalLayout class **173**
getSpecNodePositionIndex method
IlvHierarchicalLayout class **175**
getSpecRoots method
IlvTreeLayout class **102**
getSpringConstant method
IlvSpringEmbedderLayout class **78**
getStartingNode method
IlvTopologicalMeshLayout class **64**
getSubgraphs method
IlvGraphModel class **300**
getSubgraphsCount method
IlvGraphModel class **300**, **304**

getT

getTangentialOffset method
IlvLinkConnectionBoxInterface class **29**, **212**
getTipOverBranchOffset method
IlvTreeLayout class **117**
getTo method
IlvGraphModel class **300**
getTopMargin method
IlvGridLayout class **263**
getToPortIndex method
IlvHierarchicalLayout class **156**
getToPortSide method
IlvHierarchicalLayout class **155**

getV

getVerticalAlignment method
IlvBusLayout class **233**
IlvGridLayout class **260**
getVerticalGridBase method
IlvLongLinkLayout class **205**
getVerticalGridOffset method
IlvGridLayout class **262**
PerformLayout class **311**
getVerticallGridOffset method
IlvLongLinkLayout class **205**
getVerticalLinkOffset method
IlvHierarchicalLayout class **160**
getVerticalMinOffset method
IlvLongLinkLayout class **205**
getVerticalNodeLinkOffset method
IlvHierarchicalLayout class **160**
getVerticalNodeOffset method
IlvHierarchicalLayout class **160**
getVerticalOffsetToLevel method
IlvBusLayout class **235**
getVerticalOffsetToPreviousLevel method

IlvBusLayout class **235**

getW

getWestNeighbor method
IlvTreeLayout class **130**

G (continued)

global alignment parameters
Bus Layout **232**
Grid Layout **259**
Tree Layout (free mode) **106**
global end point mode parameter
Hierarchical Layout **148**
Link Layout **197**
global link style parameter
Hierarchical Layout **145**
Link Layout **194**
Tree Layout (free mode) **110**
global side association **366**
graph layout
choosing coordinate space **323**
fields of application **10**
filtering features
by graphic objects **323**
by layers **322**
for IlvGrapher **321**
in ILOG JViews **16**
nonzoomable graphic objects **324**
questions and answers **333**
reference transformers **324**
reference views **325**
specifying a reference transformer **326**
specifying the coordinates mode **327**
Graph Layout module
classes available as Beans **392**
description of **2**
features **4**
features for using layout algorithms **8**
packages **2**
graph model
concept of **298**
description of **297**
information on the geometry of the graph **300**
information on the structure of the graph **299**
modifying the geometry of the graph **301**
notification of changes of the graph **301**
retrieving data objects **302**
storing data objects **302**
grapher
attaching **18**
detaching **20**
laying out a non-JViews grapher **303**
laying out connected components of a disconnected graph **304**
laying out nested graphs **306**
grapher adapter **302**
GraphLayoutEventListener interface **294**
grid base parameter, Link Layout (long link mode) **205**
Grid Layout
alignment options **259**
applicable graph types **256**
application domains **256**
code sample **256**
description **256**
dimensional parameters **261**
features **256**
generic parameters **257**
global alignment parameters **259**
grid offset parameter **262**
incremental mode parameter **260**
individual node alignment parameter **260**
layout modes **258**
margin parameter **263**
maximum nodes per row or column parameter **260**
sample drawing **255**
specific parameters **258**
grid offset parameter
Grid Layout **262**
Link Layout (long link mode) **204**

H

Hierarchical Layout
applicable graph types **136**
application domains **136**
calculated level index parameter **183**
calculated position index parameter **184**
code sample **139**

connector style parameter **147**
crossing reduction step **138**
description **138**
end points mode parameter **148**
extremity constraints **177**
features **137**
flow direction parameter **142**
fork link shapes **156**
generic parameters **140**
global end point mode parameter **148**
global link style parameter **145**
individual end point mode parameter **149**
individual link style parameter **146**
layout constraints **169**
layout sequences **263**
level index parameter **173**
level justification parameter **143**
leveling step **138**
limitations **138**
link clipping parameter **141**, **151**
link connection box parameter **141**, **150**
link priority parameter **157**
link routing step **139**
link style parameter **144**
link width parameter **152**
node positioning step **138**
port index parameter **155**
port sides parameter **153**
position index parameter **175**
relative position constraints **175**
sample drawings **134**
side-by-side constraints **176**
spacing parameters **159**
specific parameters **142**
swim lane constraint **178**
horizontal offset parameter, Bus Layout **234**

IlvA

`IlvAnnealingLabelDescriptor` class
`getPreferredPathLocation` method **375**
`initialize` method **374**
`setTowardsPreferredPosition` method **375**
`IlvAnnealingLabelLayout` **347**
`IlvAnnealingLabelLayout` class **354**
`IlvAnnealingPointLabelDescriptor` class
constructor **361**, **362**, **363**
`IlvAnnealingPolylineLabelDescriptor` class
constructor **364**, **365**, **367**, **368**

IlvB

`IlvBusLayout` class
`getBus` method **228**
`getFlowDirection` method **230**
`getGlobalAlignment` method **232**
`getHorizontalOffset` method **235**
`getIndex` method **226**
`getLinkStyle` method **229**
`getMargin` method **235**
`getMarginOnBus` method **235**
`getMaxNumberOfNodesPerLevel` method **230**
`getOrdering` method **226**
`getVerticalAlignment` method **233**
`getVerticalOffsetToLevel` method **235**
`getVerticalOffsetToPreviousLevel` method **235**
`isBusWidthAdjustingEnabled` method **232**
provided as a Bean **392**
`setBus` method **228**
`setBusWidthAdjustingEnabled` method **231**
`setFlowDirection` method **230**, **260**
`setGlobalVerticalAlignment` method **232**
`setHorizontalOffset` method **234**
`setIncrementalMode` method **234**
`setIndex` method **226**
`setLinkStyle` method **228**
`setMargin` method **235**
`setMarginOnBus` method **235**
`setOrdering` method **226**
`setVerticalAlignment` method **233**
`setVerticalOffsetToLevel` method **235**
`setVerticalOffsetToPreviousLevel` method **235**

IlvC

`IlvCircularLayout` class
`addClusterId` method **246**
`getClusterCenter` method **251**

getClusterIds method **247**, **251**
getClusterIdsCount method **247**
getClusterNodes method **251**
getClusterRadius method **251**
getDisconnectedGraphOffset method **251**
getIndex method **246**
getLevelOffset method **251**
getLinkStyle method **252**
getOffset method **250**
getRootClusterId method **247**
isAreaMinimizationEnabled method **248**
isStarCenter method **247**
provided as a Bean **392**
removeAllClusterIds method **247**
removeClusterId method **247**
setAreaMinimizationEnabled method **248**
setClusterId method **246**
setDisconnectedGraphOffset method **251**
setLevelOffset method **251**
setLinkStyle method **252**
setOffset method **250**
setRootClusterId method **247**
setStarCenter method **247**

IlvD

IlvDefaultLabelingModel class **346**
addLayer method **379**
getLayers method **379**
removeAllLayers method **379**
removeLayer method **379**
setCoordinatesMode method **380**
setReferenceTransformer method **381**
setReferenceView method **381**
IlvDefaultLayoutProvider class
createGraphLayout method **280**, **312**
detachLayouts method **276**, **312**
getPreferredLayout method **280**, **312**
setPreferredLayout method **280**, **312**

IlvG

IlvGrapher class **303**
addNode method **341**
boundingBox method **324**
IlvGrapherAdapter class **302**
getCoordinatesMode method **328**
isLayerAdded method **322**
loadParametersFromNamedProperties method **318**, **319**
loadPreferredLayoutsToNamedProperties method **319**
removeParametersFromNamedProperties method **316**
saveParametersToNamedProperties method **316**
savePreferredLayoutsToNamedProperties method **317**
setCoordinatesMode method **327**
IlvGraphLayout class **17**
addGraphLayoutEventListener method **295**
attach method **18**
createLayoutReport method **292**
detach method **20**
getAllowedTime method **22**
getGraphLayout method **269**, **311**
getLayoutOfConnectedComponents method **305**
getLayoutRegion method **25**
getLayouts method **308**
getSeedValueForRandomGenerator method **34**
isAnimate method **23**
isFixed method **32**, **33**
isLayoutOfConnectedComponentsEnabled method **305**
isLayoutOfConnectedComponentsEnabledBy Default method **305**
isLayoutTimeElapsed method **22**
isMemorySavings method **31**
isPreserveFixedLinks method **32**
isPreserveFixedNodes method **33**
isUseDefaultParameters method **36**
isUseSeedValueForRandomGenerator method **34**
layout method **18**, **330**, **348**
layoutReport method **18**
layoutStepPerformed method **295**
PerformLayout method **311**
performLayout method **19**, **292**, **308**, **311**, **333**
setAllowedTime method **22**
setAnimate method **22**

setAutoLayout method **23**
setFixed method **32**, **33**
setLayoutOfConnectedComponents method **305**
setLayoutOfConnectedComponentsEnabled method **305**
setLayoutRegion method **25**
setLinkClipInterface method **27**, **70**, **79**, **92**, **115**, **151**, **236**, **253**
setLinkConnectionBoxInterface method **29**, **70**, **79**, **92**, **113**, **150**, **253**
setMemorySavings method **31**
setSeedValueForRandomGenerator method **34**
setUseDefaultParameters method **36**
supportsAllowedTime method **22**, **31**, **36**
supportsAnimation method **23**
supportsLayoutOfConnectedComponents method **305**
supportsLayoutRegion method **26**, **334**
supportsLinkClipping method **27**
supportsLinkConnectionBox method **29**
supportsMemorySavings method **31**
supportsPreserveFixedLinks method **32**
supportsPreserveFixedNodes method **25**, **33**, **34**
supportsRandomGenerator method **34**
supportsSaveParametersToNamedProperties method **315**
unfixAllLinks method **32**
unfixAllNodes method **33**
IlvGraphLayoutReport class
description **292**
getCode method **294**
getLayoutTime method **294**
stored information **293**
IlvGraphLayoutUtil class
IsTree static method **49**
IlvGraphModel class
addGraphModelListener method **301**
adjustmentEnd method **301**
boundingBox method **300**
createGraphModel method **300**
description of **299**
fireGraphModelEvent method **301**
getFrom method **300**
getGraphModel method **300**
getInterGraphLinks method **300**
getInterGraphLinksCount method **300**, **304**
getLinkPointAt method **300**, **304**
getLinkPoints method **300**
getLinkPointsCount method **300**
getLinks method **299**
getLinksCount method **299**, **303**
getLinksFrom method **299**
getLinksFromCount method **299**, **304**
getLinksTo method **299**
getLinksToCount method **300**, **304**
getLinkWidth method **300**
getNeighbors method **300**
getNodeDegree method **300**
getNodes method **299**
getNodesAndLinks method **299**
getNodesCount method **299**, **303**
getOpposite method **300**
getParentModel method **300**
getProperty method **302**
getRootModel method **300**
getSubgraphs method **300**
getSubgraphsCount method **300**, **304**
getTo method **300**
isInterGraphLink method **300**
isLayoutNeeded method **301**
isLink method **299**
isLinkBetween method **300**
isNode method **299**
isSubgraph method **300**
move method **301**
moveNode method **301**
removeGraphModelListener method **301**
reshapeLink method **301**
setProperty method **302**
IlvGridLayout class
getBottomMargin method **263**
getGlobalHorizontalAlignment method **259**
getGlobalVerticalAlignment method **259**
getHorizontalAlignment method **260**
getHorizontalGridOffset method **262**
getLayoutMode method **258**
getLeftMargin method **263**
getMaxNumberOfNodesPerRowOrColumn method **260**

getRightMargin method **263**
getTopMargin method **263**
getVerticalAlignment method **260**
getVerticalGridOffset method **262**
provided as a Bean **392**
setBottomMargin method **263**
setGlobalHorizontalAlignment method **259**
setGlobalVerticalAlignment method **259**
setHorizontalAlignment method **260**
setHorizontalGridOffset method **262**
setIncrementalMode method **261**
setLayoutMode method **258**
setLeftMargin method **263**
setRightMargin method **263**
setTopMargin method **263**
setVerticalAlignment method **260**
setVerticalGridOffset method **262**

IlvH

IlvHierarchicalLayout class
getCalcNodeLevelIndex method **183**
getCalcNodePositionIndex method **184**
getConnectorStyle method **148**, **156**
getDestinationPointMode method **149**
getFlowDirection method **143**
getFromPortIndex method **156**
getFromPortSide method **155**
getGlobalDestinationPointMode method **149**
getGlobalLinkStyle method **146**
getGlobalOriginPointMode method **149**
getHorizontalLinkOffset method **160**
getHorizontalNodeLinkOffset method **160**
getHorizontalNodeOffset method **160**
getLevelJustification method **144**
getLinkPriority method **158**
getLinkStyle method **146**
getNumberOfPorts method **156**
getOriginPointMode method **149**
getSpecNodeLevelIndex method **173**
getSpecNodePositionIndex method **175**
getToPortIndex method **156**
getToPortSide method **155**
getVerticalLinkOffset method **160**
getVerticalNodeLinkOffset method **160**
getVerticalNodeOffset method **160**
provided as a Bean **392**
setConnectorStyle method **147**
setDestinationPointMode method **149**
setFlowDirection method **143**
setFromPortIndex method **156**
setFromPortSide method **154**, **156**
setGlobalDestinationPointMode method **148**
setGlobalLinkStyle method **145**
setGlobalOriginPointMode method **148**
setHorizontalLinkOffset method **160**
setHorizontalNodeLinkOffset method **160**
setHorizontalNodeOffset method **160**
setIncrementalMode method **162**
setLevelJustification method **143**
setLinkPriority method **158**
setLinkStyle method **146**
setLinkWidthUsed method **153**
setNumberOfPorts method **155**
setOriginPointMode method **149**
setSpecNodeLevelIndex method **173**
setSpecNodePositionIndex method **175**
setToPortIndex method **156**
setToPortSide method **154**, **156**
setVerticalLinkOffset method **160**
setVerticalNodeLinkOffset method **160**
setVerticalNodeOffset method **160**

IlvJ

IlvJGraphLayoutProgressBar class **392**
IlvJScrollManagerView class **394**

IlvL

IlvLabelingModel
subclassing **389**
IlvLabelingModel class **346**
addLabelingModelListener method **387**
boundingBox method **386**, **390**
fireLabelingModelEvent method **387**
getLabelOverlap method **386**, **389**
getLabels method **385**, **389**
getLabelsCount method **385**
getObstacleOverlap method **386**, **390**

getObstacles method **385**, **389**
getObstaclesCount method **385**
getPolylineObstacleOverlap method **386**
getPolylinePoints method **386**
getPolylineWidth method **386**
getProperty method **388**
isBoundingBoxDependent method **387**
isLabel method **385**, **389**
isObstacle method **385**, **389**
isPolylineObstacle method **385**
method types **385**
moveLabel method **386**, **389**
removeLabelingModelListener method **387**
setProperty method **387**, **388**
IlvLabelLayout class **346**, **347**
attach method **348**
detach method **19**, **348**
performLayout method **349**
supportsSaveParametersToNamedProperties method **382**
IlvLabelLayoutReport class
method **352**
getCode method **352**
getPercentageComplete method **352**
IlvLayoutGraphicFilter class
accept method **322**
addLayer method **322**
getFilter method **322**
getLayers method **323**
getReferenceTransformer method **326**
getReferenceView method **325**
removeAllLayers method **322**
removeLayer method **322**
setFilter method **322**
setReferenceTransformer method **326**
setReferenceView method **325**
IlvLinkConnectionBoxInterface class
getBox method **29**, **208**, **211**
getTangentialOffset method **29**, **212**
IlvLinkLayout class
getDestinationPointMode method **198**
getGlobalDestinationPointMode method **197**
getGlobalLinkStyle method **195**
getGlobalOriginPointMode method **197**
getLayoutMode method **192**, **245**
getLinkStyle method **196**
getLongLinkLayout method **213**
getOriginPointMode method **197**
getShortLinkLayout method **208**
markForIncremental method **199**
provided as a Bean **392**
setDestinationPointMode method **197**
setGlobalDestinationPointMode method **197**
setGlobalLinkStyle method **194**
setGlobalOriginPointMode method **197**
setIncrementalMode method **198**
setLayoutMode method **192**, **245**
setLinkStyle method **196**
setMinFinalSegmentLength method **203**
setNodeSideFilter method **207**
setOriginPointMode method **197**
IlvLongLinkLayout class
addLineObstacle method **214**
addRectObstacle method **214**
getCalcFallbackLinks method **216**
getHorizontalGridBase method **205**
getHorizontalGridOffset method **205**
getHorizontalMinOffset method **205**
getMinFinalSegmentLength method **206**
getMinNodeCornerOffset method **206**
getVerticalGridBase method **205**
getVerticalGridOffset method **205**
getVerticalMinOffset method **205**
removeAllLinkObstacles method **214**
removeAllRectObstacles method **214**
setAllowedTimePerLink method **215**
setCrossingReductionEnabled method **215**
setFallbackRouteEnabled method **215**
setHorizontalGridBase method **205**
setHorizontalGridOffset method **204**
setHorizontalMinOffset method **205**
setMaxBacktrack method **214**
setMinFinalSegmentLength method **206**
setMinNodeCornerOffset method **206**
setNumberCrossingReductionIterations method **215**
setStraightRouteEnabled method **214**
setTerminationPointFilter method **214**
setVerticalGridBase method **205**
setVerticalGridOffset method **205**

setVerticalMinOffset method **205**

IlvM

IlvManager class
addObject method **341**
IlvManagerView class **395**
IlvMultipleLayout class **282**
getFirstGraphLayout method **287**
getLabelLayout method **287**
getSecondGraphLayout method **287**
setFirstGraphLayout method **287**
setLabelLayout method **287**
setSecondGraphLayout method **287**

IlvN

IlvNodeBoxInterface class
getBox method **207**

IlvR

IlvRandomLabelLayout class **347**, **376**
getLayoutRegion method **379**
setLayoutRegion method **379**
IlvRandomLayout class
getLinkStyle method **221**
provided as a Bean **392**
setLinkStyle method **220**

IlvS

IlvShortLinkLayout class
getGlobalSelfLinkStyle method **209**, **211**
getLinkOffset method **202**
getMinFinalSegmentLength method **203**
setAllowedNumberOfIterations method **209**
setGlobalSelfLinkStyle method **209**, **211**
setLinkOffset method **202**
setLinkToLinkCrossingPenalty method **210**
setLinkToNodeCrossingPenalty method **210**
setMinFinalSegmentLength method **203**
setSameShapeForMultipleLinks method **210**
IlvSpringEmbedderLayout class
getLinkStyle method **78**
getSpringConstant method **78**
provided as a Bean **392**
setLinkStyle method **77**
setSpringConstant method **78**

IlvT

IlvTerminationPointFilter class
getPenalty method **214**
IlvTopologicalMeshLayout class
getAllowedNodesPlacementTime method **62**
getAllowedNumberOfNodesPlacementIterations method **62**
getAllowedOptimizationTime method **62**
getExteriorCycleId method **64**
getLinkStyle method **25**, **61**
getNodesPlacementAlgorithm method **63**
getNumberOfPossibleExteriorCycles method **64**
getStartingNode method **64**
provided as a Bean **392**
setAllowedNodesPlacementTime method **62**
setAllowedNumberOfNodesPlacementIterations method **62**
setAllowedNumberOfOptimizationIterations method **62**
setAllowedOptimizationTime method **62**
setExteriorCycleId method **64**
setLinkStyle method **24**, **61**
setNodesPlacementAlgorithm method **62**
setStartingNode method **64**
IlvTopologicalMeshLayoutReport class **294**
IlvTreeLayout class
getAlignment method **108**
getAspectRatio method **123**
getBranchOffset method **117**
getCalcBackwardTreeLinks method **131**
getCalcForwardTreeLinks method **131**
getCalcNonTreeLinks method **131**
getCalcRoots method **102**
getConnectorStyle method **113**
getEastNeighbor method **130**
getFlowDirection method **106**
getGlobalAlignment method **107**
getGlobalLinkStyle method **110**

getLayoutMode method **103**
getLevelAlignment method **120**
getLinkStyle method **111**
getMaxChildrenAngle method **126**
getOrthForkPercentage method **117**
getOverlapPercentage method **118**
getParentChildOffset method **117**
getPosition method **103**
getSiblingOffset method **117**
getSpecRoots method **102**
getTipOverBranchOffset method **117**
getWestNeighbor method **130**
isRootPosition method **103**
provided as a Bean **392**
setAlignment method **108**, **109**
setAllowedTime method **127**
setAspectRatio method **123**, **126**
setBranchOffset method **116**, **123**
setConnectorStyle method **112**
setEastWestNeighboring method **130**
setFlowDirection method **105**
setGlobalAlignment method **106**, **108**
setGlobalLinkStyle method **110**
setIncrementalMode method **132**
setInvisibleRootUsed method **124**
setLayoutMode method **103**, **104**, **118**, **121**, **122**, **128**
setLayoutOfConnectedComponentsEnabled method **124**
setLevelAlignment method **120**
setLinkStyle method **111**
setMaxChildrenAngle method **126**
setOrthForkPercentage method **117**
setOverlapPercentage method **118**
setParentChildOffset method **116**, **123**
setPosition method **102**
setRoot method **102**
setRootPreference method **102**
setSiblingOffset method **116**, **123**
setTipOverBranchOffset method **117**
setWestEastNeighboring method **130**

IlvU

IlvUniformLengthEdgesLayout class
getAllowedNumberOfIterations method **87**
getLinkStyle method **87**
getPreferredLinksLength method **87**, **88**
isForceFitToLayoutRegion method **88**
isRespectNodesSizes method **88**
provided as a Bean **392**
setAllowedNumberOfIterations method **87**
setForceFitToLayoutRegion method **88**
setLinkStyle method **87**
setPreferredLinksLength method **87**
setRespectNodeSizes method **88**
IlvUniformLengthEdgesLayoutReport class **294**

I (continued)

incremental layout **16**
incremental mode parameter
Bus Layout **234**
Grid Layout **260**
Link Layout **198**
individual end point mode parameter
Hierarchical Layout **149**
Link Layout **197**
individual link style parameter
Hierarchical Layout **146**
Link Layout **195**
Tree Layout (free mode) **110**
individual node alignment parameter
Bus Layout **233**
Grid Layout **260**
Tree Layout (free mode) **107**
initialize method
IlvAnnealingLabelDescriptor class **374**
instantiating a subclass **18**
interface parameters **320**
intergraph link routing **199**
internal provider mode
Recursive Layout **267**, **269**, **273**
isAnimate method
IlvGraphLayout class **23**
isAreaMinimizationEnabled method
IlvCircularLayout class **248**
isBoundingBoxDependent method
IlvLabelingModel class **387**
isBusWidthAdjustingEnabled method

IlvBusLayout class **232**
isFixed method
IlvGraphLayout class **32**, **33**
isForceFitToLayoutRegion method
IlvUniformLengthEdgesLayout class **88**
isInterGraphLink method
IlvGraphModel class **300**
isLabel method
IlvLabelingModel class **385**, **389**
isLayerAddedmethod
IlvGrapherAdapter class **322**
isLayoutNeeded method
IlvGraphModel class **301**
isLayoutOfConnectedComponentsEnabled method
IlvGraphLayout class **305**
isLayoutOfConnectedComponentsEnabledByDefault method
IlvGraphLayout class **305**
isLayoutTimeElapsed method
IlvGraphLayout class **22**
isLink method
IlvGraphModel class **299**
isLinkBetween method
IlvGraphModel class **300**
isMemorySavings method
IlvGraphLayout class **31**
isNode method
IlvGraphModel class **299**
isObstacle method
IlvLabelingModel class **385**, **389**
isPolylineObstacle method
IlvLabelingModel class **385**
isPreserveFixedLinks method
IlvGraphLayout class **32**
isPreserveFixedNodes method
IlvGraphLayout class **33**
isRespectNodeSizes method
IlvUniformLengthEdgesLayout class **88**
isRootPosition method
IlvTreeLayout class **103**
isStarCenter method
IlvCircularLayout class **247**
isSubgraph method
IlvGraphModel class **300**
IsTree static method
IlvGraphLayoutUtil class **49**
isUseDefaultParameters method
IlvGraphLayout class **36**
isUseSeedValueForRandomGenerator method
IlvGraphLayout class **34**
IVL files **315**
compatibility issues **320**

J

Java application
sample code **342**
Java application, sample code **38**

L

label **340**
automatic placement **339**
label descriptors **359**
implementing your own **373**
Label Layout
advanced features **379**
annealing **354**
base class **347**
basic steps **341**
coordinates mode **380**
creating a new labeling model **389**
defining your own labeling model **384**
events and listeners **352**
filtering manager layers **379**
geometry of labels and obstacles **386**
getting started **340**
IlvLabelLayout parameters **353**
instantiating and attaching a subclass **348**
labeling model class **346**
loading layout parameters from IVL files **382**
nonzoomable graphic objects **380**
nonzoomable labels and obstacles **379**
notification of changes **387**
overlap calculation **386**
perfoming a layout **348**
performing a recursive layout on nested subgraphs **349**
random **376**
reference view and reference transformer **381**

releasing resources **383**
report **351**
saving layout parameters to IVL files **382**
storing and retrieving object properties **387**
structure of labels and obstacles **385**
using named properties to save layout parameters **382**
label layout report **351**
label movement policy
labels **371**
labeling model **346**
defining your own **384**
subclassing **388**
layout algorithm
basic procedure for using **38**
choosing **44**
setting the selection method **49**
table of additional information **52**
table of applicable graphs **45**
table of generic parameters supported **51**
layout constraints **169**
adding **171**
removing **171**
layout criteria
angle **15**
area **14**
bends **15**
link crossings **14**
symmetries **15**
layout listeners
Recursive Layout **279**
`layout` method
`IlvGraphLayout` class **18**, **330**, **348**
steps for implementing **330**
layout methods, types of
automatic **15**, **49**
incremental **16**
semi-automatic **49**
semiautomatic **15**
static **15**
layout modes
Grid Layout **258**
Link Layout **192**
Recursive Layout **272**
Tree Layout **103**
layout parameters
loading from IVL file **318**
saving **315**
saving to IVL file **315**
Layout Provider **311**
layout region parameter
Bus Layout **225**
Circular Layout **244**
Grid Layout **257**
Random Layout **219**
Spring Embedder Layout **76**
Topological Mesh Layout **59**
Uniform Length Edges Layout **85**
using to refine a TML graph layout **68**
layout report
class table **292**
creating **292**
elapsed time **294**
information stored in report **293**
reading **293**
using **292**
`layoutReport` method
`IlvGraphLayout` class **18**
layouts **315**
applying the same recursively **307**
Bus Layout **222**
Circular Layout **238**
defining a new type **329**
defining new type **320**
Grid Layout **255**
Hierarchical Layout **134**
Link Layout **185**
mixing different in nested graph **311**
Multiple Layout **265**, **282**
performing **19**
procedure for implementing the `layout` method **330**
Random Layout **217**
recursive **307**
releasing resources **328**
sample code for defining a new type **329**
Spring Embedder Layout **72**
Topological Mesh Layout **54**
Tree Layout **94**
Uniform Length Edges Layout **81**
`layoutStepPerformed` method
`IlvGraphLayout` class **295**

level index parameter, Hierarchical Layout **173**
level justification parameter, Hierarchical Layout **143**
level layout mode (Tree Layout)
  description **118**
  general parameters **119**
  level alignment parameter **119**
level offset parameter, Circular Layout **250**
library, ILOG JViews core library **16**
link box connection interface, Link Layout (short link mode) **211**
link categories, retrieving (Tree Layout) **130**
link clipping **27**
link clipping parameter
  Bus Layout **225**
  Circular Layout **244**, **252**
  Hierarchical Layout **141**, **151**
  Spring Embedder Layout **76**, **78**
  Topological Mesh Layout **60**, **69**
  Tree Layout **100**, **114**
  Uniform Length Edges Layout **85**, **91**
link connection box **29**
link connection box interface feature
  Link Layout **208**
link connection box parameter
  Circular Layout **244**, **253**
  Hierarchical Layout **141**, **150**
  Spring Embedder Layout **76**, **79**
  Topological Mesh Layout **60**, **70**
  Tree Layout **100**, **113**
  Uniform Length Edges Layout **86**, **92**
link crossing penalty parameter, Link Layout (short link mode) **210**
link crossings layout criteria **14**
Link Layout
  applicable graph types **186**
  application domains **187**
  bypass distance parameter (short link mode) **210**
  choosing the appropriate layout mode **193**
  code sample **190**
  description **189**
  end points mode parameter **197**
  fallback mechanism (long link mode) **215**
  features **187**
  generic parameters **190**
  global end point mode parameter **197**
  global link style parameter **194**
  grid base parameter (long link mode) **205**
  grid offset parameter (long link mode) **204**
  incremental mode parameter **198**
  individual end point mode parameter **197**
  individual link style parameter **195**
  intergraph link routing **199**
  layout mode parameter **192**
  limitations **188**
  link box connection interface (short link mode) **211**
  link connection box interface feature **208**
  link crossing penalty parameter (short link mode) **210**
  link offset parameter (short link mode) **202**
  link routing parameters (long link mode) **214**
  link style parameter **194**
  long link layout algorithm **189**
  minimal distance parameter (long link mode) **205**
  minimal node corner offset parameter (long link mode) **205**
  minimum final segment length parameter (long link mode) **206**
  minimum final segment parameter **203**
  node box interface feature **207**
  node-side filter feature **206**
  number of optimization iterations (short link mode) **209**
  obstacle parameters (long link mode) **213**
  same shape for multiple links parameter (short link mode) **209**
  sample drawing **185**
  self-link style parameter (short link mode) **208**
  short link layout algorithm **189**
  spacing parameters (long link mode) **203**
  spacing parameters (short link mode) **201**
  specific parameters **192**
  variable end point parameters (long link mode) **214**
link offset parameter, Link Layout (short link mode) **202**
link priority parameter, Hierarchical Layout **157**
link routing parameters, Link Layout (long link mode) **214**
link style parameter
  Bus Layout **228**
  Circular Layout **252**
  Hierarchical Layout **144**
  Link Layout **194**
  Random Layout **220**
  Spring Embedder Layout **77**

Topological Mesh Layout **61**
Tree Layout (free mode) **109**
Uniform Length Edges Layout **86**
link width parameter, Hierarchical Layout **152**
listener, layout event
code example **295**
description **294**
`GraphLayoutEventListener` interface **294**
listeners
label layout **352**
loading
label layout parameters from IVL file **382**
`loadParametersFromNamedProperties` method
`IlvGrapherAdapter` class **318**, **319**
`loadPreferredLayoutsToNamedProperties` method
`IlvGrapherAdapter` class **319**
local side association **366**
long link mode (Link Layout)
algorithm description **189**
fallback mechanism **215**
features **188**
grid base parameter **205**
grid offset parameter **204**
link routing parameters **214**
minimal distance parameter **205**
minimal node corner offset parameter **205**
minimum final segment length parameter **206**
obstacle parameters **213**
spacing parameters **203**
variable end point parameters **214**

M

margin on bus parameter, Bus Layout **235**
margin parameter
Bus Layout **235**
Grid Layout **263**
`markForIncremental` method
`IlvLinkLayout` class **199**
maximum nodes per level parameter
Bus Layout **230**
Grid Layout **260**
memory savings parameter
Spring Embedder Layout **76**
Topological Mesh Layout **60**
minimal distance parameter, Link Layout (long link mode) **205**
minimal node corner offset parameter, Link Layout (long link mode) **205**
minimum final segment length parameter, Link Layout (long link mode) **206**
minimum final segment parameter, Link Layout (short link mode) **203**
`move` method
`IlvGraphModel` class **301**
`moveLabel` method
`IlvLabelingModel` class **386**, **389**
`moveNode` method
`IlvGraphModel` class **301**
Multiple Layout
accessing sublayouts **287**
allowed time parameter **285**
application domain **282**
attaching graph and labeling models **287**
code sample **282**
combining multiple and recursive layout **288**
connected components parameter **285**
features **285**
for experts **287**, **288**
generic parameters **285**
percentage completion parameter **285**
recursive layout **283**
save to named properties **286**
simple layout **283**
specific parameters **286**
stop immediately parameter **286**
the reference labeling model **288**

N

nested graph **306**
Layout Provider **311**
nested subgraphs **349**
nesting for each graph in nested graph **315**
node box interface feature, Link Layout **207**
nodes
relative position constraints **169**
nodes placement algorithm, Topological Mesh Layout **62**
nodes placement allowed time, Topological Mesh Layout **62**

nodes placement iterations parameter, Topological Mesh Layout **62**
node-side filter feature, Link Layout **206**
number of iterations parameter, Uniform Length Edges Layout **87**
number of optimization iterations, Link Layout (short link mode) **209**

O

obstacle
  and nonzoomable labels **379**
  default labeling model **346**
  definition **340**
  example obstacles **341**
  examples of related obstacles **361**
  ignoring **347**
  offset parameter **370**
  positioning at **360**
  positioning at polyline **365**, **368**
  related **361**, **362**, **363**
  retrieving information on **385**, **386**
  setting immovable **347**
obstacle parameters, Link Layout (long link mode) **213**
offset parameter, Circular Layout **250**
optimization allowed time parameter, Topological Mesh Layout **62**
optimization iterations parameter, Topological Mesh Layout **62**
order of nodes parameter, Circular Layout **245**
order parameter, Bus Layout **226**
orthogonal fork percentage parameter, Tree Layout **117**
outer cycle parameter, Topological Mesh Layout
  description of **64**
outer cycle parameter, topological Mesh Layout
  using to refine a TML graph layout **68**
overlap
  calculation **386**
  value **386**
overlap percentage parameter, Tree Layout **117**

P

parameters
  description of generic parameters **20**
  generic, Annealing Label Layout **358**
  generic, Bus Layout **224**
  generic, Circular Layout **243**
  generic, Grid Layout **257**
  generic, Hierarchical Layout **140**
  generic, Link Layout **190**
  generic, Multiple Layout **285**
  generic, Random Label Layout **377**
  generic, Random Layout **219**
  generic, Recursive Layout **271**
  generic, Spring Embedder Layout **75**
  generic, Topological Mesh Layout **58**
  generic, Tree Layout **99**
  generic, Uniform Length Edges Layout **84**
  specific, Bus Layout **226**
  specific, Circular Layout **245**
  specific, Grid Layout **258**
  specific, Hierarchical Layout **142**
  specific, Link Layout **192**
  specific, Multiple Layout **286**
  specific, Random Layout **220**
  specific, Recursive Layout **277**
  specific, Spring Embedder Layout **77**
  specific, Topological Mesh Layout **61**
  specific, Tree Layout **101**
  specific, Uniform Length Edges Layout **86**
  supported by layout algorithms (table) **51**
parameters of `IlvGraphLayout`
  allowed time **22**
  animation **22**
  automatic layout **23**
  coordinates mode **24**
  description of **20**
  layout region **25**
  link clipping **27**
  link connection box **29**
  memory savings **31**
  preserved fixed links **32**
  preserved fixed nodes **32**
  random generator seed value **33**
  save parameters to named properties **34**
  use default parameters **36**
percentage completion parameter
  Hierarchical Layout **141**
  Multiple Layout **285**

Random Layout **219**
Recursive Layout **271**
Tree Layout **100**
`performLayout` method
`IlvGraphLayout` class **19**, **292**, **308**, **311**, **333**
`IlvLabelLayout` class **349**
point
positioning at **362**
point label descriptor **360**
positioning at a point **362**
positioning at an obstacle **360**
positioning on multiple criteria **363**
point labeling problem **360**
polyline label descriptor
full positioning at a link **367**
full positioning at a polyline obstacle **368**
simple positioning at a link **364**
simple positioning at a polyline obstacle **365**
global side association **366**
local side association **366**
side association **366**
port index parameter, Hierarchical Layout **155**
port sides parameter, Hierarchical Layout **153**
position index parameter, Hierarchical Layout **175**
position parameter, Tree Layout **102**
preferred layout
loading from .ivl file **319**
saving **315**
saving to IVL file **317**
preferred length parameter, Uniform Length Edges Layout **87**
preserve fixed links parameter
Bus Layout **225**
Circular Layout **244**
Hierarchical Layout **141**
Link Layout **191**
Random Layout **219**
Spring Embedder Layout **76**
Topological Mesh Layout **60**
Tree Layout **100**
Uniform Length Edges Layout **86**
preserve fixed nodes parameter
Bus Layout **225**
Circular Layout **244**
Grid Layout **258**
Hierarchical Layout **141**
Random Layout **219**
Spring Embedder Layout **76**
Topological Mesh Layout **60**
Tree Layout **101**
Uniform Length Edges Layout **86**

R

radial layout mode (Tree Layout)
adding an invisible root node **124**
alternating radial mode **122**
aspect ratio parameter **122**
description **120**
evenly spaced first circle **124**
setting a maximal children angle **125**
spacing parameters **123**
random generator seed value parameter
Random Layout **219**
Spring Embedder Layout **76**
Random Label Layout **376**
allowed time parameter **377**
code sample **377**
description **376**
features **376**
generic parameters **377**
layout region parameter **379**
percentage completion parameter **378**
random generator seed value parameter **378**
sample drawing **376**
save parameters to named properties **378**
specific parameters **378**
stop immediately parameter **378**
use default parameters **378**
Random Layout
applicable graph types **217**
code sample **218**
description **218**
features **217**
generic parameters **219**
limitations **218**
link style parameter **220**
sample drawing **217**
specific parameters **220**
reading a layout report **293**
Recursive Layout

accessing all sublayouts **275**
advanced users **277**
allowed time parameter **271**
application domain **265**
code sample, different layout styles **267**
code sample, same layout style **266**
code sample, specified layout provider **269**
combining multiple and recursive layout **288**
convenience mechanism of reference layout mode **276**
definition **266**
features **271**
generic parameters **271**
internal provider mode **267**, **269**, **273**
layout listeners **279**
layout modes **272**
layout providers for experts
layout providers
Recursive Layout **280**
percentage completion parameter **271**
reference layout mode **267**, **273**
save to named properties **272**
specific parameters **277**
specified provider mode **269**, **274**
stop immediately parameter **272**
subgraph correction **278**
recursive layout **307**
applying the same layout **307**
layout parameters **309**, **314**
mixing different in nested graph **311**
nesting for each graph in nested graph **315**
nesting several layouts for each graph **315**
simple **307**
Recursive Multiple Layout **288**
Reference Labeling Model **288**
reference layout mode
convenience mechanism (Recursive Layout) **276**
Recursive Layout **267**, **273**
reference transformer **324**, **325**, **326**
reference views **325**
refining a graph layout **66**
using fixed nodes parameter **66**
using layout region parameter **68**
using outer cycle parameter **68**
relative position constraints **169**
relative position constraints, Hierarchical Layout **175**
releasing resources **328**
`removeAllClusterIds` method
`IlvCircularLayout` class **247**
`removeAllLayers` method
`IlvDefaultLabelingModel` class **379**
`removeAllLayers`method
`IlvLayoutGraphicFilter` class **322**
`removeAllLinkObstacles` method
`IlvLongLinkLayout` class **214**
`removeAllRectObstacles` method
`IlvLongLinkLayout` class **214**
`removeClusterId` method
`IlvCircularLayout` class **247**
`removeGraphModelListener` method
`IlvGraphModel` class **301**
`removeLabelingModelListener` method
`IlvLabelingModel` class **387**
`removeLayer` method
`IlvDefaultLabelingModel` class **379**
`IlvLayoutGraphicFilter` class **322**
`removeParametersFromNamedProperties` method
`IlvGrapherAdapter` class **316**
removing
layout constraints **171**
report
classes for layout reports **292**
creating a layout report **292**
layout information **293**
reading a layout report **293**
using a layout report **292**
`reshapeLink` method
`IlvGraphModel` class **301**
resources
releasing **328**
releasing after layout **383**
respect node size parameter, Uniform Length Edges Layout **88**
respect node sizes parameter, Tree Layout **117**
ring topology, Circular Layout **241**
root clusters parameter, Circular Layout **247**
root node parameter
additional options for Tree Layout **102**
Tree Layout **101**

S

same shape for multiple links parameter, Link Layout (short link mode) **209**
save to named properties parameter
  Bus Layout **225**
  Circular Layout **245**
  Grid Layout **258**
  Hierarchical Layout **142**
  `IlvGraphLayout` **34**
  Link Layout **191**
  Multiple Layout **286**
  Random Layout **220**
  Recursive Layout **272**
  Spring Embedder Layout **77**
  Topological Mesh Layout **60**
  Tree Layout **101**
  Uniform Length Edges Layout **86**
`saveParametersToNamedProperties` method
  `IlvGrapherAdapter` class **316**
`savePreferredLayoutsToNamedProperties` method
  `IlvGrapherAdapter` class **317**
saving
  label layout parameters to IVL file **382**
  layout parameters **315**
  preferred layout **315**
self-link style parameter, Link Layout (short link mode) **208**
semi-automatic layout **49**
semiautomatic layout **15**

setA

`setAlignment` method
  `IlvTreeLayout` class **108**, **109**
`setAllowedNodesPlacementTime` method
  `IlvTopologicalMeshLayout` class **62**
`setAllowedNuamberOfIterations` method
  `IlvShortLinkLayout` class **209**
`setAllowedNumberOfIterations` method
  `IlvUniformLengthEdgesLayout` class **87**
`setAllowedNumberOfNodesPlacementIterations` method
  `IlvTopologicalMeshLayout` class **62**
`setAllowedNumberOfOptimizationIterations` method
  `IlvTopologicalMeshLayout` class **62**
`setAllowedOptimizationTime` method
  `IlvTopologicalMeshLayout` class **62**
`setAllowedTime` method
  `IlvGraphLayout` class **22**
  `IlvTreeLayout` class **127**
`setAllowedTimePerLink` method
  `IlvLongLinkLayout` class **215**
`setAnimate` method
  `IlvGraphLayout` class **22**
`setAreaMinimizationEnabled` method
  `IlvCircularLayout` class **248**
`setAspectRatio` method
  `IlvTreeLayout` class **123**, **126**
`setAutoLayout` method
  `IlvGraphLayout` class **23**

setB

`setBottomMargin` method
  `IlvGridLayout` class **263**
`setBranchOffset` method
  `IlvTreeLayout` class **116**, **123**
`setBus` method
  `IlvBusLayout` class **228**
`setBusWidthAdjustingEnabled` method
  `IlvBusLayout` class **231**

setC

`setClusterId` method
  `IlvCircularLayout` class **246**
`setConnectorStyle` method
  `IlvHierarchicalLayout` class **147**
  `IlvTreeLayout` class **112**
`setCoordinatesMode` method
  `IlvDefaultLabelingModel` class **380**
  `IlvGrapherAdapter` class **327**
`setCrossingReductionEnabled` method
  `IlvLongLinkLayout` class **215**

setD

`setDestinationPointMode` method

IlvHierarchicalLayout class **149**
IlvLinkLayout class **197**
setDisconnectedGraphOffset method
IlvCircularLayout class **251**

setE

setEastWestNeighboring method
IlvTreeLayout class **130**
setExteriorCycleId method
IlvTopologicalMeshLayout class **64**

setF

setFallbackRouteEnabled method
IlvLongLinkLayout class **215**
setFilter method
IlvLayoutGraphicFilter class **322**
setFirstGraphLayout method
IlvMultipleLayout class **287**
setFixed method
IlvGraphLayout class **32**, **33**
setFlowDirection method
IlvBusLayout class **230**, **260**
IlvHierarchicalLayout class **143**
IlvTreeLayout class **105**
setForceFitToLayoutRegion method
IlvUniformLengthEdgesLayout class **88**
setFromPortIndex method
IlvHierarchicalLayout class **156**
setFromPortSide method
IlvHierarchicalLayout class **154**, **156**

setG

setGlobalAlignment method
IlvTreeLayout class **106**, **108**
setGlobalDestinationPointMode method
IlvHierarchicalLayout class **148**
IlvLinkLayout class **197**
setGlobalHorizontalAlignment method
IlvGridLayout class **259**
setGlobalLinkStyle method
IlvHierarchicalLayout class **145**
IlvLinkLayout class **194**
IlvTreeLayout class **110**
setGlobalOriginPointMode method
IlvHierarchicalLayout class **148**
IlvLinkLayout class **197**
setGlobalSelfLinkStyle method
IlvShortLinkLayout class **209**, **211**
setGlobalVerticalAlignment method
IlvBusLayout class **232**
IlvGridLayout class **259**

setH

setHorizontalAlignment method
IlvGridLayout class **260**
setHorizontalGridBase method
IlvLongLinkLayout class **205**
setHorizontalGridOffset method
IlvGridLayout class **262**
IlvLongLinkLayout class **204**
setHorizontalLinkOffset method
IlvHierarchicalLayout class **160**
setHorizontalMinOffset method
IlvLongLinkLayout class **205**
setHorizontalNodeLinkOffset method
IlvHierarchicalLayout class **160**
setHorizontalNodeOffset method
IlvHierarchicalLayout class **160**
setHorizontalOffset method
IlvBusLayout class **234**

setI

setIncrementalMode method
IlvBusLayout class **234**
IlvGridLayout class **261**
IlvHierarchicalLayout class **162**
IlvLinkLayout class **198**
IlvTreeLayout class **132**
setIndex method
IlvBusLayout class **226**
setInvisibleRootUsed method
IlvTreeLayout class **124**

setL

setLabelLayout method
  IlvMultipleLayout class **287**
setLayoutMode method
  IlvGridLayout class **258**
  IlvLinkLayout class **192**, **245**
  IlvTreeLayout class **103**, **104**, **118**, **121**, **122**, **128**
setLayoutOfConnectedComponents method
  IlvGraphLayout class **305**
setLayoutOfConnectedComponentsEnabled method
  IlvGraphLayout class **305**
  IlvTreeLayout class **124**
setLayoutRegion method
  IlvGraphLayout class **25**
  IlvRandomLabelLayout class **379**
setLeftMargin method
  IlvGridLayout class **263**
setLevelAlignment method
  IlvTreeLayout class **120**
setLevelJustification method
  IlvHierarchicalLayout class **143**
setLevelOffset method
  IlvCircularLayout class **251**
setLinkClipInterface method
  IlvGraphLayout class **27**, **70**, **79**, **92**, **115**, **151**, **236**, **253**
setLinkConnectionBoxInterface method
  IlvGraphLayout class **70**, **79**, **92**, **113**, **150**, **253**
setLinkConnnectionBoxInterface method
  IlvGraphLayout class **29**
setLinkOffset method
  IlvShortLinkLayout class **202**
setLinkPriority method
  IlvHierarchicalLayout class **158**
setLinkStyle method
  IlvBusLayout class **228**
  IlvCircularLayout class **252**
  IlvHierarchicalLayout class **146**
  IlvLinkLayout class **196**
  IlvRandomLayout class **220**
  IlvSpringEmbedderLayout class **77**
  IlvTopologicalMeshLayout class **24**, **61**
  IlvTreeLayout class **111**
  IlvUniformLengthEdgesLayout class **87**
setLinkToLinkCrossingPenalty method
  IlvShortLinkLayout class **210**
setLinkToNodeCrossingPenalty method
  IlvShortLinkLayout class **210**
setLinkWidthUsed method
  IlvHierarchicalLayout class **153**

setM

setMargin method
  IlvBusLayout class **235**
setMarginOnBus method
  IlvBusLayout class **235**
setMaxBacktrack method
  IlvLongLinkLayout class **214**
setMaxChildrenAngle method
  IlvTreeLayout class **126**
setMemorySavings method
  IlvGraphLayout class **31**
setMinFinalSegmentLength method
  IlvLinkLayout class **203**
  IlvLongLinkLayout class **206**
  IlvShortLinkLayout class **203**
setMinNodeCornerOffset method
  IlvLongLinkLayout class **206**

setN

setNodeSideFilter method
  IlvLinkLayout class **207**
setNodesPlacementAlgorithm method
  IlvTopologicalMeshLayout class **62**
setNumberCrossingReductionInterations method
  IlvLongLinkLayout class **215**
setNumberOfPorts method
  IlvHierarchicalLayout class **155**

setO

setOffset method
  IlvCircularLayout class **250**
setOrdering method
  IlvBusLayout class **226**

setOriginPointMode method
IlvHierarchicalLayout class **149**
IlvLinkLayout class **197**
setOrthForkPercentage method
IlvTreeLayout class **117**
setOverlapPercentage method
IlvTreeLayout class **118**

setP

setParentChildOffset method
IlvTreeLayout class **116**, **123**
setPosition method
IlvTreeLayout class **102**
setPreferredLayout method
IlvDefaultLayoutProvider class **280**, **312**
setPreferredLinksLength method
IlvUniformLengthEdgesLayout class **87**
setProperty method
IlvGraphModel class **302**
IlvLabelingModel class **387**, **388**

setR

setReferenceTransformer
IlvLayoutGraphicFilter class **326**
setReferenceTransformer method
IlvDefaultLabelingModel class **381**
setReferenceView
IlvLayoutGraphicFilter class **325**
setReferenceView method
IlvDefaultLabelingModel class **381**
setRespectNodeSizes method
IlvUniformLengthEdgesLayout class **88**
setRightMargin method
IlvGridLayout class **263**
setRoot method
IlvTreeLayout class **102**
setRootClusterId method
IlvCircularLayout class **247**
setRootPreference method
IlvTreeLayout class **102**

setS

setSameShapeForMultipleLinks method
IlvShortLinkLayout class **210**
setSecondGraphLayout method
IlvMultipleLayout class **287**
setSeedValueForRandomGenerator method
IlvGraphLayout class **34**
setSiblingOffset method
IlvTreeLayout class **116**, **123**
setSpecNodeLevelIndex method
IlvHierarchicalLayout class **173**
setSpecNodePositionIndex method
IlvHierarchicalLayout class **175**
setSpringConstant method
IlvSpringEmbedderLayout class **78**
setStarCenter method
IlvCircularLayout class **247**
setStartingNode method
IlvTopologicalMeshLayout class **64**
setStraightRouteEnabled method
IlvLongLinkLayout class **214**

setT

setTerminationPointFilter method
IlvLongLinkLayout class **214**
setting a maximal children angle, Tree Layout **125**
setting even spacing for the first circle, Tree Layout **124**
setting invisible root node parameter, Tree Layout **124**
setTipOverBranchOffset method
IlvTreeLayout class **117**
setTopMargin method
IlvGridLayout class **263**
setToPortIndex method
IlvHierarchicalLayout class **156**
setToPortSide method
IlvHierarchicalLayout class **154**, **156**
setTowardsPreferredPosition method
IlvAnnealingLabelDescriptor class **375**

setU

setUseDefaultParameters method
IlvGraphLayout class **36**

setV

setVerticalAlignment method
  IlvBusLayout class **233**
  IlvGridLayout class **260**
setVerticalGridBase method
  IlvLongLinkLayout class **205**
setVerticalGridOffset method
  IlvGridLayout class **262**
  IlvLongLinkLayout class **205**
setVerticalLinkOffset method
  IlvHierarchicalLayout class **160**
setVerticalMinOffset method
  IlvLongLinkLayout class **205**
setVerticalNodeLinkOffset method
  IlvHierarchicalLayout class **160**
setVerticalNodeOffset method
  IlvHierarchicalLayout class **160**
setVerticalOffsetToLevel method
  IlvBusLayout class **235**
setVerticalOffsetToPreviousLevel method
  IlvBusLayout class **235**

setW

setWestEastNeighboring method
  IlvTreeLayout class **130**

S (continued)

short link mode (Link Layout)
  algorithm description **189**
  bypass distance parameter **210**
  features **187**
  link box connection interface **211**
  link crossing penalty parameter **210**
  link offset parameter **202**
  minimum final segment parameter **203**
  number of optimization iterations **209**
  same shape for multiple links parameter **209**
  self-link style parameter **208**
  spacing parameters **201**
side association **366**
side-by-side constraints, Hierarchical Layout **176**
simulated annealing **356**
spacing parameters
  Hierarchical Layout **159**
  Link Layout (long link mode) **203**
  orthogonal fork percentage (Tree Layout) **117**
  overlap percentage (Tree Layout) **117**
  Tree Layout (free mode) **115**, **117**
  Tree Layout (radial mode) **123**
specified provider mode
  Recursive Layout **269**, **274**
specifying a reference transformer **326**
spring constant parameter, Spring Embedder Layout **78**
Spring Embedder Layout **77**
  applicable graph types **73**
  application domains **73**
  code sample **74**
  description **74**
  features **74**
  generic parameters **75**
  limitations **74**
  link clipping parameter **76**, **78**
  link connection box parameter **76**, **79**
  sample drawings **72**
  specific parameters **77**
  spring constant parameter **78**
star center parameter, Circular Layout **247**
star topology, Circular Layout **241**
starting node parameter, Topological Mesh Layout **64**
static layout **15**
stop immediately parameter
  Bus Layout **225**
  Circular Layout **245**
  Grid Layout **258**
  Hierarchical Layout **142**
  Link Layout **192**
  Multiple Layout **286**
  Random Layout **220**
  Recursive Layout **272**
  Spring Embedder Layout **77**
  Topological Mesh Layout **60**
  Tree Layout **101**
  Uniform Length Edges Layout **86**
subgraph correction
  Recursive Layout **278**
sublayouts
  Recursive Layout **275**

supportsAllowedTime method
IlvGraphLayout class **22**, **31**, **36**
supportsAnimation method
IlvGraphLayout class **23**
supportsLayoutOfConnectedComponents method
IlvGraphLayout class **305**
supportsLayoutRegion method
IlvGraphLayout class **26**, **334**
supportsLinkClipping method
IlvGraphLayout class **27**
supportsLinkConnectionBox method
IlvGraphLayout class **29**
supportsMemorySavings method
IlvGraphLayout class **31**
supportsPrserveFixedLinks method
IlvGraphLayout class **32**
supportsPrserveFixedNodes method
IlvGraphLayout class **25**, **33**, **34**
supportsRandomGenerator method
IlvGraphLayout class **34**
supportsSaveParametersToNamedProperties method
IlvGraphLayout class **315**
IlvLabelLayout class **382**
swim lane constraint, Hierarchical Layout **178**
symmetries layout criteria **15**

T

time, stop computation algorithms **22**
tip-over alignment, Tree Layout (free mode) **108**
tip-over layout modes (Tree Layout)
aspect ratio parameter **126**
description **126**
tip leaves over **128**
tip over fast **128**
tip roots and leaves over **128**
tip roots over **128**
Topological Mesh Layout
applicable graph types **55**
application domains **55**
basic concepts **57**
code sample **58**
description of algorithm **56**
features **55**
generic parameters **58**
limitations **56**
link clipping parameter **60**, **69**
link connection box parameter **60**, **70**
link style parameter **61**
nodes placement algorithm **62**
nodes placement allowed time parameter **62**
nodes placement iterations parameter **62**
optimization allowed time parameter **62**
optimization iterations parameter **62**
outer cycle parameter **64**
refining a layout **66**
sample drawings **54**
specific parameters **61**
starting node parameter **64**
using fixed nodes parameter to refine **66**
using layout region parameter to refine **68**
using outer cycle parameter to refine **68**
Tree Layout
adding an invisible root node (radial mode) **124**
algorithm description **97**
alternating radial mode **122**
applicable graph types **95**
application domains **95**
aspect ratio parameter (tip-over mode) **126**
aspect ration parameter **122**
calculating link shapes **98**
calculating node positions **98**
calculating the spanning tree **97**
code sample **98**
compass directions **103**
connector style parameter **111**
evenly spaced first circle (radial mode) **124**
features **95**
flow direction parameter **104**
free layout mode **104**
generic parameters **99**
global link style parameter **110**
individual link style parameter **110**
interactive editing **132**
layout modes **103**
level alignment parameter **119**
level layout mode **118**
limitations **96**
link clipping parameter **100**, **114**

link connection box parameter **100**, **113**
link style parameter **109**
making incremental changes **132**
orthogonal fork percentage **117**
overlap percentage parameter **117**
position parameter **102**
radial layout mode **120**
respect node sizes **117**
retrieving link categories **130**
retrieving list of root nodes used by algorithm **102**
retrieving list of specified root nodes **102**
root node parameter **101**
root node parameter options **102**
sample drawings **94**
setting a maximal children angle (radial mode) **125**
setting a root node **101**
spacing parameters (free mode) **115**
spacing parameters (radial mode) **123**
specific parameters **101**
specifying east-west neighbors **129**
specifying root node preference **102**
specifying the order of children **132**
tip-over alignment (free mode) **108**
tip-over layout modes **126**

U

`unfixAllLinks` method
`IlvGraphLayout` class **32**
`unfixAllNodes` method
`IlvGraphLayout` class **33**
Uniform Length Edges Layout
additional node repulsion weight **89**
applicable graph types **83**
application domains **83**
code sample **84**
description **84**
features **83**
force fit to layout region parameter **88**
generic parameters **84**
limitations **83**
link clipping parameter **85**, **91**
link connection box parameter **86**, **92**
link length weight **89**
link style parameter **86**
maximum allowed move per iteration **88**
node distance threshold **91**
number of iterations parameter **87**
preferred length parameter **87**
respect node size parameter **88**
sample drawings **81**
specific parameters **86**

V

variable end point parameters, Link Layout (long link mode) **214**
vertical offset parameter, Bus Layout **235**
vertical offset to previous level parameter, Bus Layout **235**

W

width adjusting
Bus Layout **231**